

# Designing the Architecture of Clotho 3.0: An Interactive Pattern Story

**Ernst Oberortner, Swapnil Bhatia,  
Douglas Densmore**  
Department of Electrical and Computer  
Engineering  
Boston University  
{ernstl,swapnilb,dougd}@bu.edu

**J. Christopher Anderson**  
University of California Berkeley  
jcanderson@berkeley.edu

## Abstract

The Clotho software platform is tailored for the synthetic biology domain and offers a rich set of Apps to engineer novel biological systems. Clotho 2.0 has several shortcomings regarding its architecture and data object model. In this paper, we tell an interactive pattern story about the design process of the Clotho 3.0 architecture in order to overcome the drawbacks of Clotho 2.0, as well as to address newly emerging functional and non-functional requirements. The pattern story reflects our design decisions and solutions to prepare the Clotho 3.0 architecture for a pattern-based architecture review.

## 1. Introduction

Synthetic biologists engineer complex and novel living systems, which do not exist in nature. This domain incorporates large amounts of biological data, such as the compositions of large Deoxyribonucleic acid (DNA) sequences and meta-data information, for example, information about the fabrication lab, literature information, assembly algorithms and trees, or bio-safety features. Automated DNA fabrication methods, as well as software and computational tools have gained momentum in the synthetic biology research domain. It is challenging to develop user-friendly software tools that can keep up with the tremendous increase of biological data.

The Clotho software platform [XBB+11,DDJ+09] is tailored to manage large amounts of synthetic biological data in relational data repositories. Clotho 2.0 offers a rich repertoire of applications (Apps) that allow its users to create, query, modify, and operate on the biological data located in the repositories. Since all Clotho components must reside currently at one user's local machine, we have identified the following shortcomings: (1) the users' software environments are uncontrollable, (2) a versioning of each user's local Clotho installation is not supported, (3) data inconsistencies among several data repositories are difficult to manage. Also, Clotho's current architecture makes it difficult for the App developers to extend the data object model for user- or lab-specific requirements, as well as to integrate the new App into an existing Clotho installation.

To overcome these shortcomings, we have designed a new architecture for Clotho 3.0. In this paper, we prepare the new architecture for a Pattern-based Architecture Review (PBAR) [HA11] by telling an interactive pattern story. The story serves as a vehicle to demystify the

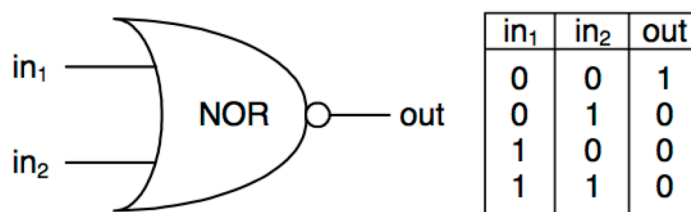
architectural design decisions taken during the design process [TA05]. Our story's patterns stem from various pattern books, such as the Design Patterns [GHJ+95] book and the POSA series [BMR+06, SSR+00, POSA4]. The storytelling is based on the stories told in [BHS07b], and [SP09]. Ultimately, we utilize a Data Flow View [AZ05] to illustrate the Clotho 3.0 architecture and its operating data object model. However, we do not present any technical details about the architecture's implementation, the data exchange, and the data storage.

This paper is structured as follows: The next section, Section 2, gives basic background of the synthetic biology domain. In Section 3, we explain Clotho 2.0's architecture and its drawbacks, and we list the requirements on the new architecture. In Section 4 we tell the interactive pattern story about the architecture's design process and how the architecture tackles its requirements. Then, in Section 5 we present the resulting Clotho 3.0 architecture and data object model. The paper concludes and presents future work in Section 6.

## 2. What is Synthetic Biology?

Synthetic biologists try to construct novel biological organisms, which do not exist in nature and behave in a predictable and controllable way. Therefore, well-engineered biological components, such as promoters, ribosome binding sites, and genes are synthesized [BS05, KC10]. Deoxyribonucleic acid (DNA) serves as the storage molecule of biological information for the construction and function of cells. DNA is interpreted to build a protein through a two-step process: transcription and translation. Transcription is the process during which mRNA (messenger ribonucleic acid) is transcribed from DNA by RNA-polymerase. Translation is the process of translating the transcribed mRNA to a protein, by using a ribosome molecule according to the genetic code [TT03].

Synthetic biologists try to construct novel strands of DNA that produce proteins that behave in a predictable way [End05]. The transcription and translation processes include various functioning and well-engineered genetic parts. For example, a promoter part initiates the DNA transcription and a terminator terminates the DNA transcription. Transcription factors enhance or repress promoters. A ribosome-binding site (RBS) is an un-translated sequence important for the translation process. Such parts are of particular importance in synthetic biology in order to construct devices that behave in a controlled and predicted way.

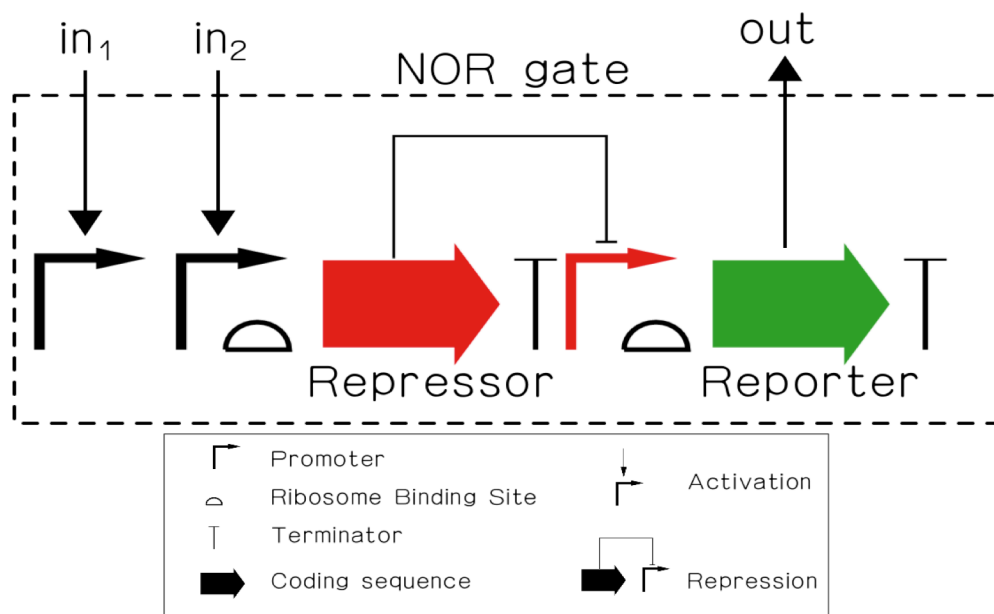


**Figure 1:** Logical NOR Gate

To enhance the understandability of synthetic biology, we draw an analogy between logical gates and genetic devices. A logical NOR gate (see **Figure 1**) has two input signals and one

output signal that depends on the two input signals. The output signal is “on” only if both input signals are “off”. In an equivalent way it is possible to control – turn “on” or “off” – the expression of genes.

Similarly, a genetic NOR gate [Voi06] (see **Figure 2**) is a device that contains several genetic parts of DNA. It starts with two inducible promoter parts, which initiate – dependent on their input signals – an expression of a reporting gene (Reporter), such as a green fluorescent protein (GFP). Input signals are environmental signaling molecules that can chemically bind to the promoters. If both input signals are “on” or one of them, then the repressor part will repress the red downstream promoter, and ultimately the reporting fluorescent protein will not glow green. If both input signals do not bind to the inducible promoters -- the input signals are “off” -- the repressor part allows the red downstream promoter part to initiate the reporter part, making ultimately the green fluorescent protein glow.



**Figure 2:** Genetic NOR Gate

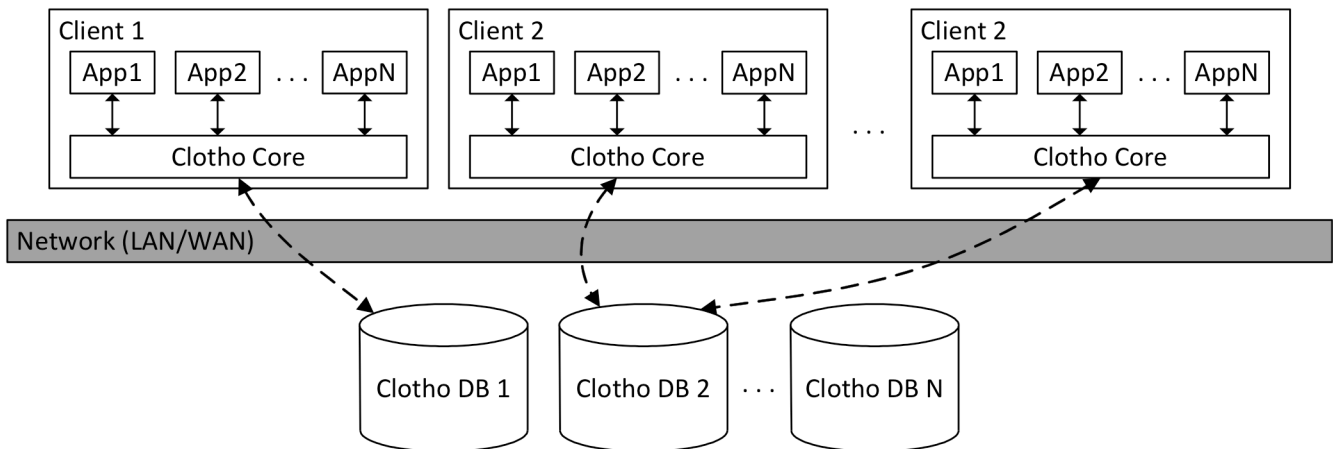
Synthetic biology encounters various interesting applications, such as the production of biofuels, the creation of tumor killing bacteria, or the fabrication of biosensors. For biosensors, for example, genetic gates can be wired together into more complex circuits in order to detect, for example, arsenic in water. Software and computational tools facilitate the specification and design of novel genetic systems, as well as to automate the physical assembly of DNA using liquid handling robotics. Also, software tools tailored for the synthetic biology domain need to scale appropriately to manage the increasing amount of novel biological data.

### 3. The Clotho 2.0 Architecture, its Shortcomings, and Requirements

Now, we explain the Clotho 2.0 architecture and its drawbacks, followed by a description of requirements for the Clotho 3.0 architecture. In this section, as well as in the rest of the paper, we differentiate between Clotho users and Clotho clients. A Clotho **user** is an individual, such as a synthetic biologist, who utilizes Clotho and its Apps to design a novel biological system, for example, to cure cancer. Therefore, the Clotho user utilizes a Clotho **client**, which is Clotho's software component that offers various Apps to the Clotho user in order to achieve their work.

#### 3.1 The Clotho 2.0 Architecture

In Clotho 2.0, the Clotho Core and the user-required Apps are installed at each user's local machine. The Clotho Core builds the interface between the Apps and a Clotho database and is responsible to retrieve and store the Apps' required data. From a pattern perspective, the Clotho 2.0 architecture (see **Figure 3**) recalls the `REPOSITORY` [Shaw96] and the `SHARED REPOSITORY` [VKZ04] architectural patterns.



**Figure 3:** The Clotho 2.0 Architecture

However, Clotho 2.0 has several shortcomings. Every Clotho Core can have an additional local database, and the Clotho user decides at Clotho's startup either to connect to the local or to the remote database. However, it is not possible to synchronize the local database's data with the data in a specified remote database. If the user decides to work on the local database, then there is no way to push the local database's data into a remote database. Also, a pull mechanism is missing to fetch data from a remote database into the local database. If a Clotho user utilizes, for example, an App to fetch data from the connected remote database and the requested data is not available, then an empty data set will be returned. However, it is possible that the requested data resides in another remote data repository. The loose coupling between the local and the remote data repositories, as well as among the remote repositories, can result into serious data inconsistencies.

Since it is difficult to define a data model that fits all kinds of Apps and users, Clotho 2.0 offers a rich data model to store biological data, such as composite parts of DNA sequences, as well as meta-information, such as fabrication lab or bio-safety information. From an App developer's perspective, it can be difficult to squeeze the App's data into Clotho 2.0's data model, since the data model does not offer extension points.

### **3.2 Requirements on the Clotho 3.0 Architecture**

In this section we list the requirements of Clotho 3.0 that the new architecture should satisfy. We differentiate between functional requirements (*FR*) and non-functional requirements (*NFR*) [BMR+06]. First, we explain the non-functional requirements on Clotho 3.0:

#### ***NFR-1: Lighten the strong coupling between the Apps and the Clotho Core***

In Clotho 2.0, the Clotho Core and the user's Apps must be installed together on one machine. Hence, the Apps and the Clotho Core are tightly coupled, which leads to the aforementioned drawbacks (see Section 3.1). We envision a physical separation that is transparent to the Clotho users to lighten the strong coupling between the Apps and the Clotho Core.

#### ***NFR-2: Testability and Debugging Facilities***

The Clotho 3.0 architecture should be modular in order to provide testing and debugging facilities of common synthetic biological data flows. Such a data flow consists, for example, of the retrieval of existing DNA parts, their synthesis into a complex system, and the final storage of the new biological system into a data repository.

#### ***NFR-3: Scalability***

Due to the desired physical separation of the Clotho Core and the Apps, the Clotho 3.0 architecture should support simultaneous communications between multiple Cores and multiple Apps. Hence, the Clotho Core must be scalable and any performance drops should not influence the Apps' performance.

#### ***NFR-4: Quick data access and retrieval***

Clotho's main application is to manage large amounts of biological data. Since data retrieval, data modification, and data storage are subject to automation processes, Clotho 3.0 should support an efficient retrieval and storage of the data.

Besides the non-functional requirements, Clotho 3.0 must also fulfill the following functional requirements:

#### ***FR-1: Support of various types of clients and requests***

Due to the desired separation between the Clotho Core and the Apps, Clotho 3.0 should support multiple types of clients that host the Apps, such as lightweight clients, web-based clients, or mobile clients.

The entire Clotho 2.0 infrastructure does not have any type of clients because the entire Clotho software package is installed on one user's local machine. But, Clotho 3.0 should provide a solution to integrate existing Clotho 2.0 clients seamlessly.

Because of the physical separation of the Apps and the Clotho Core, both have to communicate with each other over a network. Therefore, various technologies should be supported, such as web sockets or XML HTTP requests.

***FR-3: Maintaining the state of the client's and the user's***

Because various types of clients should be supported, it should be possible to keep track of the clients' state. For example, a Clotho user logs into Clotho to design a novel biological system. Because the design process will not be finished within a normal working day, the user should be able to resume the design process at any later time. Therefore, Clotho 3.0 should provide session handling.

***FR-4: Asynchronous data exchange***

Clotho users should be able to continue their work while they are waiting until the Clotho Core responds the requested data. Clients should not stall in the meantime, and hence, asynchronous data exchange between the Clotho clients and the Clotho core must be supported.

***FR-5: Notify users about relevant data changes***

Clotho 2.0 offers an "Auto Update" feature, making it possible to pull updated software versions from the Clotho software repository, such as the Clotho Core or of a particular App. For Clotho 3.0 it is desired to notify the Clotho users automatically if there are not only new software versions available, but also new biological data, such as novel parts of DNA sequences.

***FR-6: Support of management facilities***

Clotho 2.0 does not offer any kind of management facilities, such as to monitor the status of the data repositories or to configure and install pre-packaged Clotho App solutions for users. However, in Clotho 3.0 we envision user-friendly management interfaces that are accessible, for example, over the web.

***FR-7: Extensible Data Object Model***

The data object model of Clotho 2.0 is rich, however, it does not provide any extension points to customize it for new Apps, as well as for individual Clotho users and/or laboratories. In Clotho 3.0 we must improve this major shortcoming of Clotho 2.0.

## 4. The Interactive Pattern Story

In this section, the paper's main contribution, we tell an interactive pattern story about the design process of Clotho's new architecture and its data object model. The story reflects the taken design decisions in order to fulfill the aforementioned requirements. We recall the problem and solution statements of the story's pattern in the Appendix.

Every step in the story ends with choices how to proceed reading the story. It is your decision how you want to read the story. Our story offers three options to start:

**Option 1:** *If you are interested in the design steps of the Clotho 3.0 architecture, then start reading the story with Step 1.*

**Option 2:** *If you are interested in the architecture's data object model, then start with Step 7.*

**Option 3:** *If you don't want to experience our design process, then you can see the resulting architecture and its data object model in Section 5.*

### Step 1: Lighten the coupling between the Apps and the Clotho Core

In Clotho 2.0, the Apps and the Clotho Core are tightly coupled and installed together on a Clotho user's local machine. We started to think about “How can we lighten the strong coupling between the Apps and the Clotho Core?” (*NFR-1*).

Since we had already a separation between the Application Logic (i.e. the Apps) and the Data Access Logic (i.e. the Clotho Core) we thought about “Why do we not separate the Apps and the Clotho Core physically?” The Apps can send requests to the Clotho Core in order to fetch or store biological data. Hence, we decided in favor of a `CLIENT/SERVER` [AZ05] architectural style. The separation into a client and a server component improves the testing and debugging facilities (*NFR-2*), since both components can be tested and debugged separately.

**Option 1:** *If you want to know how we designed the Clotho Core as a server component, then go to Step 2.*

**Option 2:** *If you are interested how Clotho 2.0 clients can get extended in order to access the Clotho 3.0 server, then go to Step 6.*

### Step 2: Design of the Clotho Core

As a next step, we discussed about “How should we design and structure the Clotho Core in order to fulfill the requirements as good as possible?”

In order to answer this question, we first defined the duties and responsibilities of the new Clotho core. We agreed on three inter-related responsibilities: (1) handle to communication between the clients and the Clotho core, (2) do the processing of the clients' requests, and (3) access the data repositories. As a result, the new Clotho Core consists of three `LAYERS` [BMR+06]: the **Communication Layer** (Layer I), the **Request Handling Layer** (Layer II), and the **Data Access Layer** (Layer III).

Following the `LAYERS` pattern helps to satisfy the *NFR-2* requirement – testability and debugging facilities – in two ways. First, every layer and its consisting components can be tested individually. Second, a layered architecture eases to follow the `INDIRECTION LAYER` pattern [AZ05, Zdu03] in order to trace user requests to detect the origin of eventual bugs and system failures.

***Option 1:** If you want to know how the Communication Layer was being designed, keep on reading with Step 3.*

***Option 2:** If you want to know more about the design of the Request Handling Layer, then go to Step 4.*

***Option 3:** If you are interested in the Data Access Layer, you can skip to Step 5.*

### **Step 3: The Clotho Core's Communication Layer**

The Communication Layer of the new Clotho architecture receives requests from the clients and responds to the clients accordingly. One of the first questions that arose during our design sessions, were “How can the Communication Layer support various types of clients and request formats (FR-1)?”

We decided to deploy a **Router** component that forwards the incoming requests depending on their types. This solution recalls the `MESSAGE ROUTER` pattern [HW03]. Regarding the `MESSAGE ROUTER` pattern, every request gets forwarded to another component via a `MESSAGE CHANNEL`. But, to which component does our Router forward the incoming requests? We introduced a **Listener** component that serves every type of request that the Clotho Core supports. Adding a new request type requires, however, to re-configure and extend the Message Router and to deploy a new Listener. To add new request types, we plan on utilizing `PLUGIN` patterns [Mar99].

The Listener component forwards the request to an appropriate component in the Request Handling Layer, which does the actual request processing. This separation recalls the `FORWARD-RECEIVER` pattern [BMR+06]. The request forwarding from the Communication Layer to the Request Handling Layer resembles the `INDIRECTION LAYER` [Zdu03] architectural pattern.

***Option 1:** If you are interested in the further components of the Communication Layer to achieve Session Handling (FR-3) and Asynchronous Data Exchange (FR-4), then keep on reading.*

***Option 2:** If you want to get more information about the Request Handling Layer, then go to Step 4.*

***Option 3:** If you are interested in the Data Access Layer, then skip to Step 5.*

#### **Step 3.1: Session Handling Support**

Although Clotho offers various Apps to support synthetic biologists in the design of novel biological systems, the design process is tedious. Similar to software architectures, novel biological systems are not fully designed within one design session. Hence, one further question was “How should the Clotho 3.0 architecture support session handling?”



We had a look into the pattern literature and discovered a paper on session patterns [Sor02]. Especially the `KEEP SESSION AT THE SERVER` pattern got our attention. We have integrated a **Mind** component into the Communication Layer that stores the session information of every Clotho user, making it possible to satisfy the requirement *FR-3*. However, we do not present any further technical details how the Mind stores the session data in this manuscript. In the Clotho 3.0 architecture, the **Router** component requests the Mind component to get the user's current session data.

***Option 1:** Go to the next step (Step 3.2) if you want to know how we augmented the Communication Layer to support asynchronous data exchange.*

***Option 2:** If you are interested in the Request Handling Layer's components, then go to Step 4.*

***Option 3:** If you want to experience the components of the Data Access Layer now, then go to Step 5.*

### **Step 3.2: Support of Asynchronous Data Exchange**

Whenever a Clotho client sends a request to the Clotho server, the client should not stall. For example, the verification of a designed synthetic biological system against bio-safety and biophysical requirements, can take a long time. In such a case, Clotho users should be able to continue their work while they are waiting for the server's response. How does the Clotho 3.0 architecture support asynchronous data exchange?

We had a look into the pattern literature and discovered a book about Remoting Patterns [VKZ04]. In this book, the authors explain patterns for an asynchronous data exchange. To fulfill the requirement *FR-4*, we integrated a **Callback-Handler** component into the Communication Layer, which follows the `RESULT-CALLBACK` pattern to support asynchronous data exchange. The Callback-Handler is responsible to respond the requested data to the clients. The Listener component of the Communication Layer instantiates an appropriate Callback-Handler by following the `FACTORY METHOD` pattern [GHJ+95] and forwards a reference to the instantiated Callback-Handler to the Request Handling Layer.

Now, you have experienced the Communication Layer of the Clotho 3.0 architecture. It consists of four inter-related components, i.e. the Router, the Listener, the Mind, and the Callback-Handler.

***Option 1:** In the next step (Step 4), we explain the design of the Request Handling Layer and its components.*

***Option 2:** If you are, however, interested in the Data Access Layer, which builds upon the Request Handling Layer, then go to Step 5.*

### **Step 4: The Clotho Core's Request Handling Layer**

In one of our design sessions we came across the design decision about how the actual process handling should be handled. Regarding the stated requirements, the Clotho architecture should be able to process requests of multiple clients simultaneously (*NFR-3*). The ultimate question was “How can we build a scalable request processing architecture?”

Again, in the pattern literature we found advice how to achieve a well-established solution. The data exchange between client and server follows the `COMMAND PROCESSOR` pattern [BMR+06] that builds on the `COMMAND` pattern [GHJ+95]. In the Request Handling layer we integrated an **Executor** component, which receives the incoming request objects from the Listener component of the Communication Layer. The Executor unwraps the requests' data objects and instantiates an **Assistant** following the `FACTORY METHOD` pattern [GHJ+95], which perform the actual processing of the data objects. An Assistant calls the Data Access Layer to access the data stored in the data repositories. To return the result to the requesting client, the Assistant invokes the assigned Callback-Handler.

Ultimately, the Clotho 3.0 architecture achieves load balancing – according to the `LOAD BALANCER` pattern [Sor02] – via four collaborating components: the Router and Listener components of the Communication Layer, and the Executor and Assistant components in the Request Handling Layer. The collaboration of the Listener and the Executor resembles the `FORWARD-RECEIVER` pattern [BMR+06].

***Option 1:** To experience the Data Access Layer and its components keep on reading with the next step (Step 5).*

***Option 2:** If you are interested in the integration of the management facilities into the architecture, then go to Step 6.*

***Option 3:** If you want to discover the data object model now, then skip to Step 7.*

### **Step 5: The Clotho Core's Data Access Layer**

We dedicated some of our design sessions to a come up with appropriate data access and storage solutions. In these design decisions we were stating questions about “How should the Clotho 3.0 architecture be designed to satisfy the requirement regarding efficient data access (*NFR-4*)?”

The Data Access Layer of the Clotho 3.0 architecture builds on the Request Handling and consists of components to store and retrieve the in the client request specified data objects. The **Collator** component provides an API to the Request Processing Layer to fetch and retrieve data objects from the data repositories. The Collator fetches the requested data objects first from the **Collector**, which is a cache the holds frequently used objects in-memory for efficient data retrieval. If the requested data is not in the Collector (i.e. the cache), the Collator must query the data from the **Persistor** component, which is responsible for the physical data retrieval from the appropriate data repositories.

If the data objects are serialized to flat files, the Collator also manages and organizes the indexing mechanisms. The **Collator** component assembles three patterns:

- An `EXPLICIT INTERFACE` [BH03] that provides an API to store, to retrieve, and to manage data in the data repositories.
- An `OBJECT MANAGER` [BH03] to checks the clients' session information in order to protect the data from illegal access, and
- A `CACHE ACCESSOR` [Noc03] to separate the caching logic from data access details.

Since the **Collector** component is a cache to keep frequently accessed data objects in memory for efficient data retrieval, the **Collector** can be seen as a CONTAINER [BHS07a].

The **Persistor** component is an EXPLICIT INTERFACE since it provides an interface to a physical storage subsystem to store and fetch data objects physically from the data repositories. The **Persistor** also follows the HOST ACCESS patterns (Host Communication Agent; Flat File Write), as well as the SERIALIZER pattern [RSB+98].

Now, you know the layered architecture of the Clotho core, how the layers are build on each other, and how each layer's components interact with each other.

***Option 1:** If you want to learn how Clotho 3.0 utilizes patterns to support Clotho 2.0 clients, then keep on reading with the next step (Step 6).*

***Option 2:** If you are interested in the augmentation of the architecture with management facilities, then skip to Step 7.*

***Option 3:** If you want to experience the extensible data object model of Clotho 3.0, then go to Step 8.*

### **Step 6: Support of Clotho 2.0 Clients**

One of the biggest challenges that we were facing is “How can Clotho 2.0 clients be integrated into the Clotho 3.0 infrastructure?” Which well-proven solutions do exists in order to overcome this problem?

After having a look into the pattern literature we came up with two possible solutions: Utilizing the BROKER pattern and/or the PROXY pattern [BMR+06]. Both are effective solutions to integrate Clotho 2.0 clients into a Clotho 3.0 infrastructure.

***Option 1:** If you want to know about the architecture's management facilities, then keep on reading with Step 7.*

***Option 2:** Go to Step 8 now if you want to experience Clotho's new data object model.*

***Option 3:** If you want to stop reading the story now, then you can go to Section 5, in which we visualize and exemplify the resulting architecture and its data object model.*

### **Step 7: Integration of Management Facilities**

After we have had designed the architecture's layers, their components, and their interactions we started to discuss about addressing FR-6. How should we provide management interfaces and pre-packaged App solutions to the Clotho users?

As a result, each Clotho server is equipped with a **Management** component to maintain and monitor all resources and components of the corresponding server. In our design session, we decided that the architecture's management facilities should also be responsible to notify users about relevant data changes (FR-5). Therefore, we follow the PUBLISHER-SUBSCRIBER pattern [BMR+06]. First, the Clotho users must specify about which updates they want to be

notified. This information can be stored at the server, for example, in the user's *Certificate* object. The **Manager** component can be utilized (1) to subscribe and unsubscribe users and clients to and from specific topics and (2) to manage topics. The **Mind** component of the Communication Layer stores the data updates to the user's sessions and, hence, can notify the users appropriately.

The **Manager** component is also connected to the **App Store**, a `CONTAINER` [BHS07a] of all available Clotho Apps and libraries. Apps are lightweight containers that encode a list of Libraries. Libraries are the portable version of the data objects held by components of the Data Access Layer, i.e., **Collator**, **Collector**, and **Persistor**. Apps represent `PLUG-INS` and the App Store follows the `PLUG-IN REGISTRATION` pattern [Mar99].

Clotho administrators can handle the installation, removal, and update of Apps for specific Clotho users via the Manager component. Clotho Apps encode a list of Libraries, which are portable data objects held by the **Collator**, the **Collector**, and the **Persistor**. If you go into the App Store and browse it as a developer, you are likely looking for libraries. If you browse it as a user, you see Apps, which are finished tools. When a Clotho Core is asked to install an App, it sees what of that list of Libraries it is missing, downloads and adds them.

***Option 1:** If you are interested now in the design of Clotho's data object model, then keep on reading with Step 8.*

***Option 2:** If you are interested in a graphical representation of the architecture, then continue reading with Section 5.*

### **Step 8: Clotho 3.0's Data Object Model**

Since Clotho 2.0 offers a rich data object model already, we did not want to change it completely. We just wanted to modify it in a way that it is easy to extend and customizable (*FR-7*). We started to discuss about “How can we achieve this”?

In the pattern literature we endeavored patterns how to design the data object model for being extensible. The *Datum* class is the root of the data object model, which is an `EXTENSION INTERFACE` [SSR+00]. Following the `OBJECT IDENTIFIER` pattern [Kel98], the *Datum* class holds an `UUID` attribute, making it possible that every data object has a unique identifier.

For the time being, the Clotho data object model provides some pre-defined sub-classes, i.e. *User*, *Doo*, and *Certificate*. The *User* classes deal with user management issues. In order to detect failures, data flows – tracked by the *Doo* objects – can be traced/logged through the entire architecture in every component, making it also possible to debug Clotho's architecture easily (*NFR-2*). *Certificate* classes deal with user management and authentication mechanisms.

The *Datum* class can be extended individually with user-required functionalities. However, introducing new Datum sub-classes requires the implementation a new Assistant that is able to process the objects of the newly introduced Datum subclass. As explained in Step 4, Assistants reside in the Request Handling Layer, and the architecture's implementation follows the `PLUGIN` patterns [Mar99] in order to integrate new Assistants easily. In order to store individual *Datum* objects in the data repositories, the *ObjBase* class must be extended

and a reference to a specific *Schema* object must be provided. Instances of the *Connection* class have, for example, scripts that handle the communication with the Communicator component of the Clotho core.

Also, the *DataField* class is an EXTENSION INTERFACE [SSR+00] that wraps primitive types, such as Strings (cf. WRAPPER patterns [GHJ+95]).

**Option 1:** *If you jumped directly into Step 7, then you go to Step 1 now to inform yourself about the design of the Clotho 3.0 architecture, its layers, and their collaborating components.*

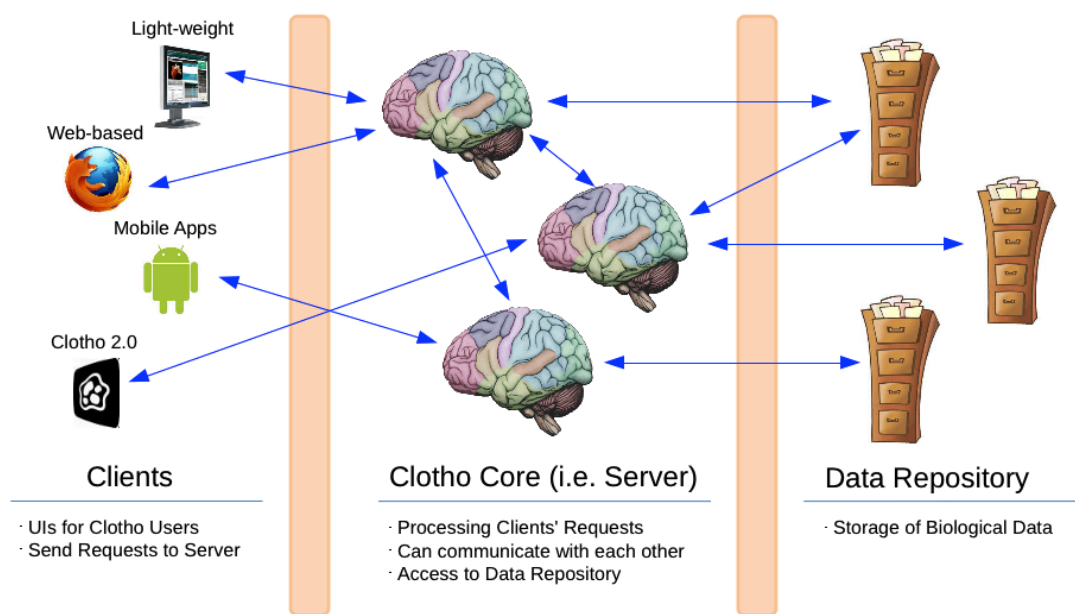
**Option 2:** *If you are already aware of the architecture's layers and its components, then you reached the end of the story. You should now have a look into the resulting architecture in Section 5.*

## 5. The Clotho 3.0 Architecture and Data Object Model

In this section, we present the outcomes of our design sessions, the Clotho 3.0 architecture and its data object model. First, we present the architecture from a high-level perspective, followed by an exemplified Data Flow View through the Clotho Core using a typical workflow of the synthetic biology domain. The section concludes with the illustration of Clotho's new data object model.

### 5.1 A High-Level Overview of Clotho's new Architecture

In **Figure 4** we give a high-level overview of the Clotho 3.0 architecture, where the clients, the Clotho Core, and the data repositories are physically separated. However, the Clotho core and the data repositories must not be physically separated necessarily. Clotho 3.0 supports four types of clients: web-based applications, lightweight clients, mobile clients, and – to maintain backward compatibility – Clotho 2.0 clients. Every server is connected to at least one data repository, which is a database that contains the biological data corresponding to Clotho 3.0's data object model.

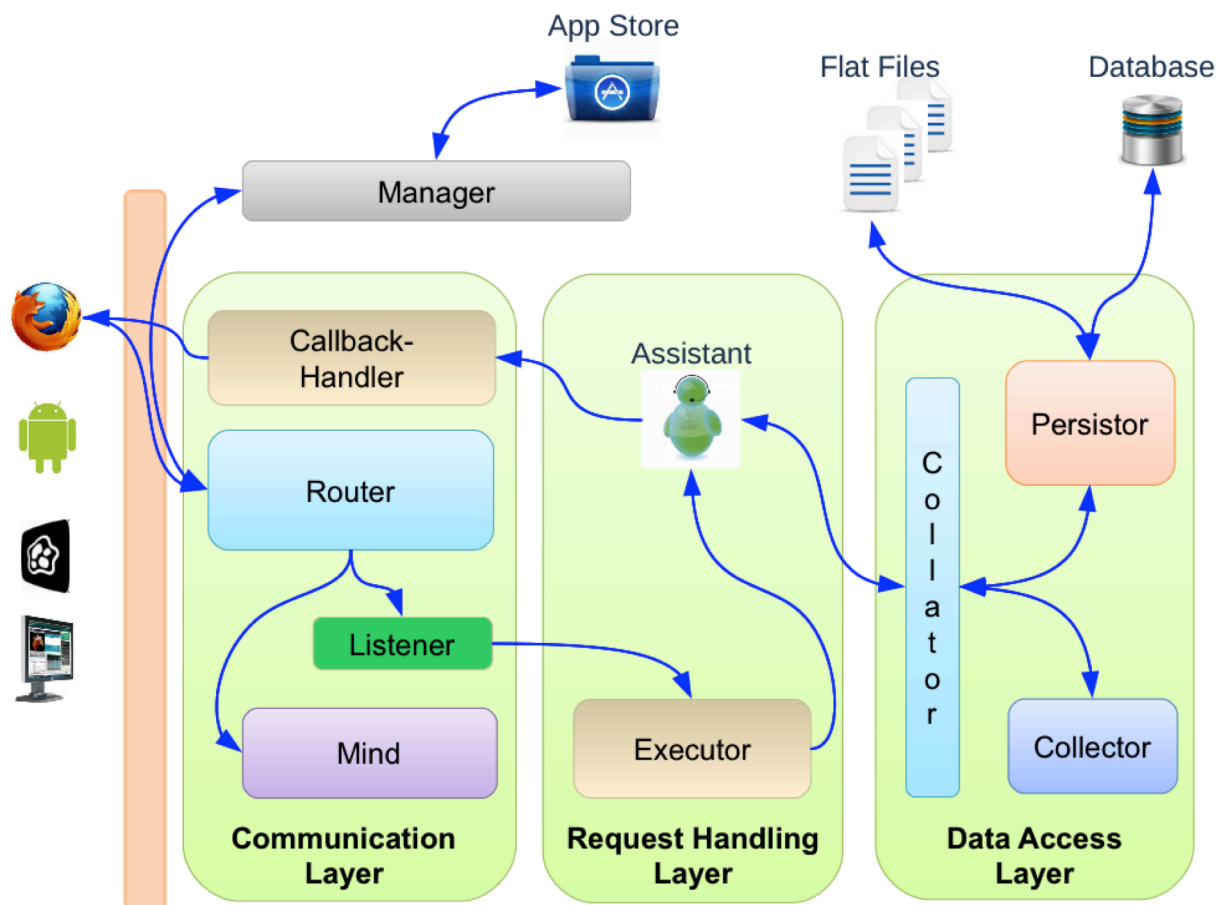


**Figure 4:** A High-Level Overview of the Clotho 3.0 Architecture

For example, to request biological data from a data repository, the client sends an appropriate request to the server (i.e. the Clotho Core). The server processes the clients' requests, performs the in the request specified operations on the data repository, and returns the results back to the requesting clients.

## 5.2 A Data Flow View of the Clotho Core

In **Figure 5** we illustrate how data flows through the layers of the Clotho Core using a typical synthetic biological workflow. In this scenario, the Clotho user utilizes the EugeneScripter App to write a Eugene script [BCW+11] to constrain the composition of parts of DNA sequences.



**Figure 5:** A Data Flow View of the architecture's layers and collaborating components

At the client side, the EugeneScripter App connects automatically to the server Clotho core at startup. For authentication purposes, the **Mind** component evaluates the client's certificate and creates a new session if the certificate is valid and no session is active. After successful authentication, the user can now utilize the App, for example, to request stored DNA sequences. The App initiates the underlying middleware (not shown in **Figure 5**), which marshals and transmits the requests to the Clotho server.

At the server-side, the **Router** component of the Communication Layer retrieves the encapsulated request, unwraps it, and – dependent on the connection type – instantiates an appropriate **Listener**. Then, the Forwarder creates a **Callback-Handler** with the client and request information in order to return the requested data accordingly. The **Listener** forwards the request's data to an appropriate **Executor** of the Request Handling Layer, which itself

instantiates an **Assistant**. The Assistant does the actual processing of the client request and it receives a reference to the previous instantiated Callback-Handler.

In order to request the data repositories, the Assistant invokes an appropriate method of the **Collator** component, which builds the interface between the Request Handling Layer and the Data Access Layer. First, the Collator queries the requested data from the **Collector** component, which has the functionality of a cache. If the data is present in the Collector, then the Collector returns the data to the requesting Collator. If the data is not present in the Collector, then the Collator invokes the **Persistor** component. The Persistor queries a data repository for the requested data and returns the results to the Collator.

To return the requested data to the client, the Collator returns the data to the Assistant which invokes the, to the request associated, Callback-Handler. At the client-side, the Clotho core forwards the received data to the EugeneScripter App, which ultimately displays the data to the user for further processing.

Not involved in this workflow are the **Manager** component and the **App Store**. For example, the Manager is utilized to manage the users' certificates to guarantee that only authorized users can log into the system.

### 5.3 Clotho's new Data Object Model

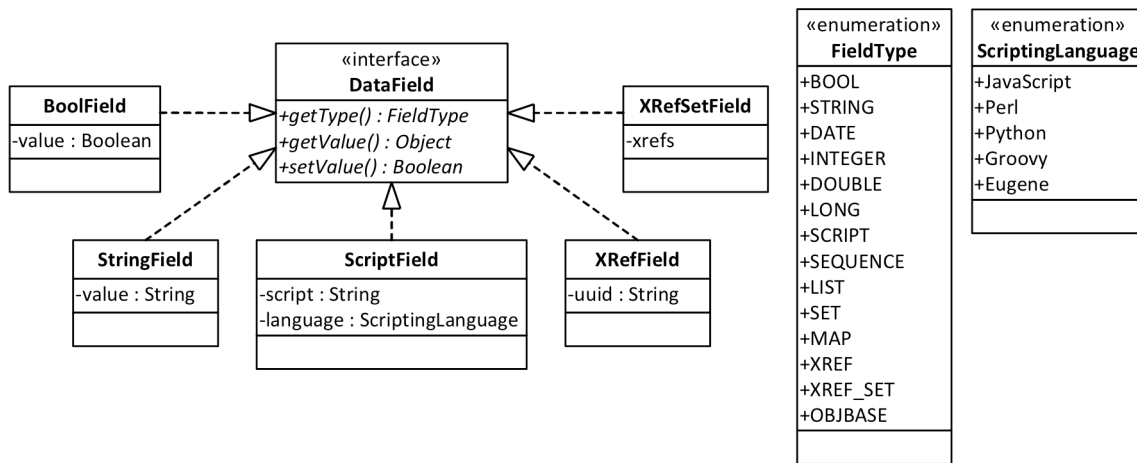
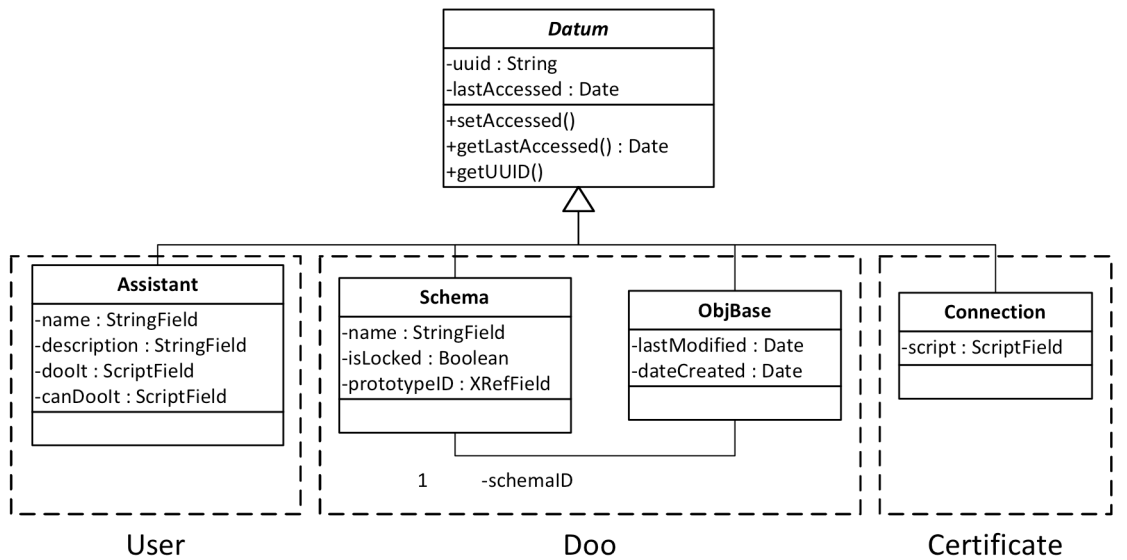
In **Figure 6** we sketch Clotho's data object model. The *Datum* class is the base class for objects that can be persisted into data repositories. The Datum class is a lightweight bean implementation to facilitate persistence and memory management. Each *Datum* has a universal unique identifier (UUID), making every object stored in the data repositories unique.

The group of *User* classes represents information about individual Clotho users and can store information, for example, when a user logged off or on.

In the Clotho data model, a *Doo* is a unit of work that a client initiates, making it possible to manage, schedules, log, and track workflows. A Doo guarantees that the action requested doesn't just disappear when it fails – it is tracked during execution to confirm that it successfully completes. If something goes awry, a stack of Doos can be sent to the development team in order to fix bugs. The name 'Doo' comes from modeling work in terms of asynchronous execution of discrete tasks that need to be 'done on', 'done by', or 'are due on'. Originally, a Doo is a *Wrapper* [GHJ+95] around a bolus of work that would be implemented by an Assistant at some time scheduled by the Executor.

In the lower portion of **Figure 6**, we present the Primitives of the Clotho data model. Primitives implement the *DataField* interface and are `WRAPPERS` [GHJ+95] around various low-level data types, such as Strings, numbers, or lists. For example, a BoolField can only accept a data change if it can be interpreted as being a boolean. Currently, there are a finite number of data fields, but, additional data fields, such as an image data file, can be added easily.





**Figure 6:** The Data Object Model of Clotho 3.0

Every *ObjBase* in the data model holds a reference to a *Schema* object. *Schemas* can be recursively built-up from the *DataField* wrappers to handle low-level data validations. For example, the *Schema* class can be instantiated with a 'Institution' *DataField*, where 'UC Berkeley' or 'Boston University' is a *StringField*, representing its name. The 'Institution' *DataField* is then referenced by an *ObjBase* object, which holds information about the creation and last modification date. Since all Strings will ultimately get compressed to a *StringField*, there are many ways to index a String's content to facilitate meaningful queries in the **Collator** component.

Certificate objects deal with user rights and data access permissions. Every Clotho user is equipped with a certificate, hold and evaluated by the **Mind**. The **Manager** component can be utilized to create, modify, and delete users and its certificates.

## 6. Summary and Future Work

In this paper, we told an interactive pattern story about the design process of the Clotho 3.0 architecture. The new architecture follows a CLIENT/SERVER architectural style and tackles several drawbacks of the Clotho 2.0 architecture. The story reflects the design decision that we were facing in various design session in order to create an architecture that fulfills several functional and non-functional requirements. In the story, we align the architecture's layers and components with patterns from the pattern literature. After telling the story, we visualize the architecture using a data flow view, based on a typical workflow of the synthetic biology domain. The main idea is to illustrate the Clotho 3.0 architecture and the taken architectural design decisions to pattern experts for a Pattern-based Architecture Review (PBAR).

As future work, we focus on communication solutions among multiple Clotho Cores. From a synthetic biological standpoint, this is of interest because one Clotho Core can contact other Clotho Cores if some requested biological data is not available. However, this is particularly challenging because storing novel biological designs into one data repository can lead to data inconsistencies with data that already exists in another data repository.

## Acknowledgements

We gratefully thank our shepherd, Stefan Sobernig, who gave us constructive feedback to improve the paper's quality. It was primarily Stefan's idea to tell the interactive pattern story.

## References

- [AZ05] P. Avgeriou and U. Zdun, Architectural Patterns Revisited – A Pattern Language In: *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, Irsee, Germany (2005)
- [BCW+11] L. Bilitchenko, A. Liu, S. Cheung, E. Weeding, B. Xia, et al. (2011) Eugene – A Domain Specific Language for Specifying and Constraining Synthetic Biological Parts, Devices, and Systems. *PLoS ONE* 6(4): e18882. DOI=10.1371/journal.pone.0018882
- [BH03] F. Buschmann, K. Henney, Explicit Interface and Object Manager, *EuroPLOP* 2003.
- [BHS07a] F. Buschmann, K. Henney, D.C. Schmidt, Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing, *Wiley*, 2007
- [BHS07b] F. Buschmann, K. Henney, D.C. Schmidt, Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages, *Wiley*, 2007
- [BMR+06] F. Buschmann; R. Meunier; H. Rohnert; P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns, *John Wiley & Sons*, 1996
- [BS05] S. A. Benner and A. M. Sismour. Synthetic biology. *Nature Reviews Genetics*, 6(1), 2005. <http://www.ncbi.nlm.nih.gov/pubmed/15995697>
- [DDJ+09] D. Densmore, A. Van Devender, M. Johnson, and N. Sritanyaratana, A platform-based design environment for synthetic biological systems. In *The Fifth Richard*

- Tapia Celebration of Diversity in Computing Conference: Intellect, Initiatives, Insight, and Innovations (TAPIA'09)*. ACM, New York, NY, USA, 24–29. DOI=10.1145/1565799.1565806
- [End05] D. Endy. Foundations for engineering biology. *Nature*, 438 (7067):449–453, 2005. ISSN 1476-4687. doi: 10.1038/nature04342.
- [GHJ+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.
- [HA11] N. Harrison, P. Avgeriou, Pattern-Based Architecture Reviews, *IEEE Software*, vol.28, no.6, pp.66-71, Nov.-Dec. 2011, DOI=10.1109/MS.2010.156
- [HW03] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
- [Kel98] W. Keller, Object/Relational Access Layers – A Roadmap, Missing Links and More Patterns, In Proceedings of the *Third European Conference on Pattern Languages of Programming and Computing (EuroPLoP)*, 1998.
- [KC10] A. S. Khalil and J. J. Collins, Synthetic biology: applications come of age, *Nature Reviews Genetics* 11, 367-379 (May 2010). doi:10.1038/nrg2775
- [Mar99] K. Marquardt, Patterns for Plug-Ins, In Proceedings of the *Fourth European Conference on Pattern Languages of Programming and Computing (EuroPLoP)*, 1999
- [Noc03] C. Nock, Data Access Patterns: Database Interactions in Object-Oriented Applications, *Addison-Wesley Professional*, 2003. ISBN 0-13-140157-2.
- [RSB+98] D. Riehle, W. Siberski, D. Baeumer, D. Megert, H. Zuellighoven, Serializer, in *Pattern Languages of Program Design 3*, Addison-Wesley, 1998, Chapter 17, 293–312.
- [SC97] M. Shaw and P. C. Clements, A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In Proceedings of the *21st International Computer Software and Applications Conference (COMPSAC)*, pages 6–13. IEEE Computer Society, 1997.
- [Sor02] K. E. Sorensen, Session Patterns, EuroPlop 2002.
- [SP09] J. Siddle and M. Platts, "Choose your own architecture" – Interactive Pattern Storytelling. In *Transactions on Pattern Languages of Programming II*, James Noble and Ralph Johnson (Eds.). Springer-Verlag, Berlin, Heidelberg 16–33.
- [SSR+00] D. Schmidt, M. Stal, H. Robert, F. Buschmann. *Pattern Oriented Software Architecture - Volume 2: Patterns for Concurrent and Networked Objects*. ISBN: 0-471-60695 2, Wiley & Sons, 2000.
- [TA05] J. Tyree and A. Akerman, Architecture Decisions: Demystifying Architecture, *IEEE Software*, vol.22, no.2, pp. 19- 27, March-April 2005 doi: 10.1109/MS.2005.27
- [TT03] N. J. Trun and J. E. Trempy, *Fundamental Bacterial Genetics*. Blackwell Science, 2003.
- [VKZ04] M. Völter, M. Kircher, U. Zdun, Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware, *Wiley Series in Software Design Patterns*, ISBN: 0470856629, Wiley & Sons, Oct 2004
- [Voi06] C. A. Voigt, Genetic parts to program bacteria. *Current Opinion in Biotechnology*, 17(5):548–557, 2006.
- [XBB+11] B. Xia, S. Bhatia, B. Bubenheim, M. Dadgar, D. Densmore, and J. C. Anderson,

Clotho v2.0: A Software Platform for the Creation of Synthetic Biological Systems, Methods in Enzymology, Volume 498, 2011.

[Zdu03]

U. Zdun, Patterns of Tracing Software Structures and Dependencies, EuroPLOP 2003.

## Appendix: Related Patterns

LAYERS [BMR+06]	Imagine that you are designing a system whose dominant characteristic is a mix of low- and high-level issues, where high-level operations rely on the lower-level ones.	Structure your system into an appropriate number of layers and place them on top of each other.
MESSAGE ROUTER [HW03]	How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?	Insert a special filter, a Message Router, which consumes a Message from one Message Channel and republishes it to a different Message Channel depending on a set of conditions.
BROKER [BMR+06]	When distributed components communicate with each other, some means of inter-process communication is required. If components handle communication themselves, the resulting system faces several dependencies and limitations.	Clients access the functionality of servers by sending requests via a BROKER. A broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.
PROXY [BMR+06]	We do not want to hard-code its physical location into clients, and direct and unrestricted access to the component may be inefficient or even insecure. Additional control mechanisms are needed.	Let clients communicate with the data repositories via a representative rather than the data repositories itself. A PROXY offers interfaces of the data repositories and allows additional pre- and post-processing, such as data access control.
COMMAND PROCESSOR [BMR+06]	An application that includes a large set of features benefits from a well-structured solution for mapping its interface to its internal functionality. You often need to implement services that go beyond the core functionality of the system for the execution of user requests.	The Command Processor pattern builds on the COMMAND design pattern [GHJ+95]. Both patterns follow the idea of encapsulating requests into objects. Whenever a user calls a specific function of the application, the request is turned into a command object. The COMMAND PROCESSOR pattern illustrates more specifically how command objects are managed.
FORWARD-RECEIVER [BMR+06]	A common way to build distributed applications is to make use of available low-level mechanisms for inter-process communication (IPC). These often introduce dependencies on the underlying operating system and network protocols.	Distributed peers collaborate to solve a particular problem. A peer may act as a client, requesting services, as a server, providing services, or both. The details of the underlying IPC mechanism for sending or receiving messages are hidden from the peers by encapsulating all system-specific functionality into separate components.
PUBLISHER-SUBSCRIBER [BMR+06]	A situation often arises in which data changes in one place, but many other components depend on this data. We are looking for a more general change-propagation mechanism that is applicable in many contexts. When some internal data element changes all clients that depend on this data have to be updated.	One dedicated component takes the role of the publisher (SUBJECT [GHJ+95]). All components dependent on changes in the publisher are its subscribers (OBSERVER [GHJ+95]). The publisher maintains a registry of currently subscribed components. Whenever the publisher changes state, it sends a notification to all

		its subscribers. Whenever a client wants to become a subscriber, it uses the subscribe interface offered by the publisher. Analogously, it can unsubscribe.
CONTAINER [BHS07a]	Components implement self-contained business or infrastructure logic that can be used to compose applications. Since components may be deployed across a diverse range of applications and platforms, however, they cannot assume specific execution scenarios and technical environments.	Define a container to provide the execution environment for a component that supports the necessary technical infrastructure to integrate components into application-specific usage scenarios, and on specific system platforms, without tightly coupling the components with the applications or platforms.
EXTENSION INTERFACE [SSR+00]	It is hard to design stable interfaces, because requirements can change in unanticipated ways after components have been delivered and integrated into applications. When not handled carefully, these changes can break existing client code that uses the components.	Program clients to access components via separate interfaces, one for each role a component plays, rather than programming clients to use a single component that merges all its roles into a single interface or implementation. In detail: Export component functionality via extension interfaces, one for each semantically related set of operations. Every component must implement at least one extension interface.
EXPLICIT INTERFACE [BH03]	A component represents an implementation of a self-contained unit of functionality and deployment with a published usage protocol. Clients can use a component as a building block in providing their own functionality. However, direct access to the full component implementation would lead to clients depending on the component internals, which ultimately increases an application's internal coupling.	Separate the interface of a component from its implementation so that the latter can be modified transparently and independently. Export the interface to the clients of the component, but keep its implementation and location private. A call from the client through this explicit interface will be to the component, but the client code will depend only on the interface and not on the component implementation.
OBJECT MANAGER [BH03]	Certain kinds of objects within an application – in particular server-side components, system resources, and singletons – require access control and a managed lifecycle. It is otherwise hard to maintain and use them efficiently, correctly, and without degrading the application's quality of service. However, implementing such functionality within the objects themselves overloads them with peripheral responsibilities and makes their simple and uniform use harder rather than simpler.	Separate object usage from object lifecycle and access control. Introduce a separate object manager whose sole responsibility is to manage and maintain exclusively a given set of objects. Clients can use the object manager to gain access to objects with specific capabilities.
DECORATOR / WRAPPER [GHJ+95]	Sometimes we want to add responsibilities to individual objects, not to an entire class. One way to add responsibilities is with inheritance. Inheriting attributes and methods from	A more flexible approach is to enclose the component in another object that adds appropriate attributes and methods. The enclosing object is called a decorator. A decorator conforms to the interface of the

	another class puts the inherited attributes and methods into every subclass instance. This is inflexible, however, because the choice of attributes and methods is made statically.	component it decorates so that its presence is transparent to the component's clients.
FACTORY METHOD [GHJ+95]	Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
RESULT-CALLBACK [VKZ04]	The client needs to be informed actively about results of asynchronously invoked operations on a remote object. That is, if the result becomes available to the REQUESTOR, the client wants to be informed immediately, so that it can react on the availability of the result. In the meantime the client executes concurrently	Provide a callback-based interface for remote invocations on the client. Upon an invocation, the client passes a RESULT CALLBACK object to the REQUESTOR. The invocation returns immediately after sending the invocation to the server. When the result is available, the distributed object middleware invokes a predefined operation on the RESULT CALLBACK object, passing it the result of the invocation.
INDIRECTION LAYER [Zdu03]	Trace information can consist of information in electronic documents (such as the source code), but also of information derived from dynamic invocation data (and data flows). It is hard to integrate these two kinds of trace information, as the former is statically provided in the sources, whereas the later is obtained from the running system. How to gather and integrate all relevant static and dynamic trace information in a unique way?	Provide an INDIRECTION LAYER between the application logic and the instruction set of the (sub-)system that should be traced. The general term "instruction set" can refer to a whole programming language, but it can also refer to the public interface of a used component, sub-system, or layer. The INDIRECTION LAYER wraps all accesses to the relevant (sub-) system and should not be bypassed. In this INDIRECTION LAYER provide custom hooks to extract the relevant trace information.
KEEP SESSION AT THE SERVER [Sor02]	Session specific data has to be stored in between requests, and made available to the code handling a request.	Keep all session specific data on the server. Keeping all data on the server and making sure it will never leave the server, means you have no need to write elaborate error checking code to validate data every time it reenters the system from the client. It also frees you from implementing code that converts from the form the data is stored in while in the server (e.g. hierarchies of objects) to a form that can be transmitted over the wire between client and server.
LOAD BALANCER [Sor02]	How do you distribute the load from multiple users accessing your system over several instances of the system? How can you take a server instance down for maintenance without disrupting user requests? How can you ensure undisturbed service to users if one of your	A Load Balancer is a system that all requests have to pass through on their way to the server. The job of the Load Balancer is to direct the requests to the server instance that should handle them.

	multiple server instances crashes?	
CACHE ACCESSOR [Noc03]	Data access operations consume a significant portion of an enterprise system's resources. They are a common source of performance bottlenecks, so optimization efforts often focus on implementing data access components as efficiently as possible.	Decouple caching logic from the data model and data access details.
SERIALIZER [RSB+98]	Every major application needs to read objects from and write them to a varying number of backends with different representation formats. Application classes should have no knowledge about the external representation format, which is used to represent their instances. Otherwise, introducing a new representation format or changing an old one would require changing almost every class in the whole system.	Therefore, every application class provides an interface called Serializable. This interface consists of two methods, one for reading and one for writing the object. Subclasses of the Serializable interface implement this interface by accepting Reader/Writer objects and by reading from or writing their attributes to them.
PLUG-IN [Mar99]	An application that is required to be highly adaptable, or be extensible to support future functionality or modules. How can functionality be added late? How can the functionality be increased after shipping?	Factor out functionality, and place it in a separate component that is activated at run time. This component is called a PLUG-IN.
PLUG-IN-REGISTRATION [Mar99]	Application has defined Framework Interfaces and Plug-In Definitions. Plug-Ins are available. User or application decides at run time which Plug-In to activate. How are the Plug-Ins known to the application?	The application defines a place where it looks for available Plug-Ins. Each Plug-In installs itself there.
HOST ACCESS [Kel98]	How do you connect an object/relational access layer to a host computer running a transaction system?	Write all queries to a communication agent, using bundled write. Install another communication agent on your host computer that unpacks the query packets and executes them one by one under the control of the host transaction monitor. Send back a packet containing query results or the return codes of the access layer modules from the host computer.
OBJECT IDENTIFIER [Kel98]	How do you represent an object's individuality in a relational database?	Assign the objects a synthetic key that accompanies the object from birth to destruction. Bury the key with the object.