# Architecture Description Document

*Adamant Software Framework*

## 1 Purpose

Adamant is a component-based, model-driven framework designed for constructing reliable and reusable embedded, real-time software systems. The purpose of this document is to discuss the architecture of the Adamant framework. This document is written at a high level, where an in depth knowledge of programming is not required. The intended audience for this document is managers and systems engineers looking to understand what an Adamant system looks like and software engineers interested in using Adamant to develop their own software systems.

## 2 Introduction

The purpose of the Adamant framework is to improve the way embedded software is designed, implemented, tested, and communicated when compared to traditional development practices. The conventional pattern for developing embedded software usually employs a “clone and own" methodology from one project to another. The use of heritage design and high code reuse is often acclaimed when progressing to a new project from a previous one. However, the process of copying and modifying source for new projects is not only labor intensive, it is error prone. While the code might exhibit some semblance of reuse, it usually needs to be modified to interface with its new execution context, and it always needs to be retested and revalidated against new requirements. Lean development costs are often presented in the early phases of the project, but expenses inevitably increase in the face of changing requirements during the testing and integration of the system as it nears delivery. The Adamant framework addresses these issues by providing mechanisms for direct, unmodified code reuse of common core components from project to project. Project specific requirements can be compartmentalized in a collection of independent *components* with well defined interfaces. Should requirements change on future projects, these non-core components can easily be replaced by components adhering to the same interface, but with different internal behavior. The use of components also eases unit testing and simulation, as the interface for either a test harness or a simulation harness is inherently known.

Historically, embedded software development practices dictate a requirements and design first, implement and test later philosophy. While this top down approach attempts to ensure that implementation does not begin until the design is well understood, changing requirements throughout the life cycle of a project is inevitable. This can result in an implementation that is not reflected in the final design, hampering communication and productive review of the architecture. The Adamant framework enforces consistency between the design and implementation of the system through the use of *models*. The models not only produce a well constructed visual language for viewing the architecture, that can be understood by software engineers and systems engineers alike, they also generate the scaffolding code for the system, ensuring that the visual diagrams are truthfully reflected in the implementation.

By dividing the architecture into a collection of independent, model-backed components, the Adamant framework aims to improve the reliability, reusability, adaptability, and testability of embedded software development.

# 3 Concepts

At its core, Adamant is a component-based framework. Component-based frameworks approach application design as a set of reusable functional or logical entities called *components*, which expose well-defined interfaces for inter-component communication. The design of components follow some key principles:

- **Encapsulation** - A component hides the details of its internal behavior and state and only allows other components to access its functionality through the interface it presents.
- **Independence** - Components do not depend on other components, they depend only on their interfaces. This means that changes to one component are unlikely to ripple through the entire system.
- **Reusability** - Some components may be designed for a project-specific task, however, many components can be designed in a generic way such that they can be reused in similar systems with no modification. In addition, a component can be placed in a variety of execution contexts and be still expected to function as designed.
- **Replaceability** - Because a component is simply represented by its interfaces, it can be substituted for another component as long as it adheres to the same interface.

Components in Adamant communicate with one another through strongly typed *connectors*. Components can have many connectors, of many different types, which can either send or receive data or both. Because connectors are exposed in the design, data flow within an Adamant architecture is not obscured, but rather can be easily inspected by a viewer of the design.

The Adamant framework is also model-driven. Users of Adamant model components and their connections using a simple text language. This language is not only used to generate graphical views of the project architecture, but it also generates the code which makes up the backbone of the architecture. The behavior of these components is then hand-coded by a developer who fills in a set of specific methods, stubbed out by the code generator. Later, when the design needs to change, the developer can modify the model and have these changes immediately reflected in the software. In this way, the design of a project's architecture, as crafted into the model, is explicitly tied to the actual software implementation. This inherent transparency between model and implementation is powerful during all phases of software development from initial design, through integration and test.

The following sections describe key concepts of the Adamant framework.

## 3.1 Connectors

Connectors are the primary means by which components communicate with one another. Their main purpose is to provide a gateway for data transfer while removing any dependencies one component might have on another. In a traditional software system, if one module wants to call a function in another module, it will need to include that other module, thus declaring a dependency on it. This coupling can lead to software maintenance problems down the road. Later, when requirements change or a new project comes along, one of these modules may need to be changed. Depending on how robust the interface is between the two modules, the two pieces of code may need to be manually untangled, carefully modified in concert, or even rewritten from scratch. In order to facilitate robust interfaces while removing dependencies, Adamant uses connectors. In this paradigm, if one module needs to call a function in another module, it does so through a connector. The modules do not depend on one another. Instead, they both depend on the connector between them. Thus, changes to one module will no longer affect the other module, so long as they both agree on the definition of their shared connector.

Connectors have a few important properties. The first is the connector's *type*. The type of a connector determines the datatypes that can be passed along it. Data passage along a connector is said to occur when that connector is *invoked*. Upon connector invocation, an associated function is called which implements the *behavior* of the connector, usually acting on the transferred data in some way. The *invoker* of a connector is defined as the component which initiates the behavior, while the *invokee* component is defined as the component which is responsible for performing the behavior. In Adamant, two types can be passed along the connector: the *invocation type*, or simply *type*, and the *return_ type*. A connector's *type* is always passed from the *invoker* to the *invokee*, while the *return_ type* is always passed from the *invokee* to the *invoker.* Each of these types may be arbitrarily complex using compound datatypes such as arrays or structures. Predefined basic types, such as integers or floats, as well as pointers, are also acceptable connector types.

Making the distinction between *invoker* and *invokee* is important when describing the direction and timing of the data flow between components. Because connectors can specify both a *type* and/or a *return_ type*, the direction of data flow can be in the corresponding or opposite direction of the invocation direction. The relationship between the invocation direction and data flow direction is formalized by the connector's *kind.* Unlike connector *types*, which refer to a connector's datatype, connector *kinds* are used to classify the direction and synchroneity of data flow along a connector. Connector kinds fall into one of four categories, whose names correspond to the direction of the data flow with respect to the *invokee.*

1. **In Connectors** - One way connectors which simply send data from the *invoker* component to the *invokee* component. These connectors support both synchronous as well as asynchronous communications.
2. **Return Connectors** - One way connectors in which no data is sent from the *invoker* component to the *invokee* component, however data is synchronously returned from the *invokee* to the *invoker.* Only synchronous communication is possible with this category.
3. **In Return Connectors** - Two way connectors in which data is sent from the *invoker* component to the *invokee* component and separate return data is returned from the *invokee* back to the *invoker.* Only synchronous communication is possible with this category.
4. **In Out Connectors** - Two way connectors in which modifiable data is sent from the *invoker* component to the *invokee* component. The data can be modified by the *invokee* before being synchronously returned to the *invoker.* Only synchronous communication is possible with this category.

Note that asynchronous communication is not possible with *In Return Connectors*, *Out Connectors*, or *In Out Connectors* because data flows in the opposite direction of the connector invocation. Because data is returned to the *invoker* during the invocation, the communication is, by definition, synchronous. However, two-way, asynchronous data flow is possible between components by using two *In Connectors* invoked in opposite directions.

A name is given to the *invoker* and *invokee* side of each connector category called a *kind.* The connector kind pairs that make up each category are shown below:

| **Category** | **Invoker Kind** | **Invokee Kind** | **Has *type*?** | **Has *return_ type*?** |
|---|---|---|---|---|
| In | `send` | `recv_sync` or `recv_async` | yes | no |
| Out | `get` | `return` | no | yes |
| In Return | `request` | `service` | yes | yes |
| In Out | `provide` | `modify` | yes | no |

In this way, compatible connector pairs can be identified. For instance if one component has a `send` connector of type $T$, this can only be connected to another component if that component exposes a connector of type $T$ with a kind of `recv_sync` or `recv_async`. Connector kinds from different categories cannot be used together.

Each compatible connector kind has a unique graphical representation to improve the clarity of component diagrams.

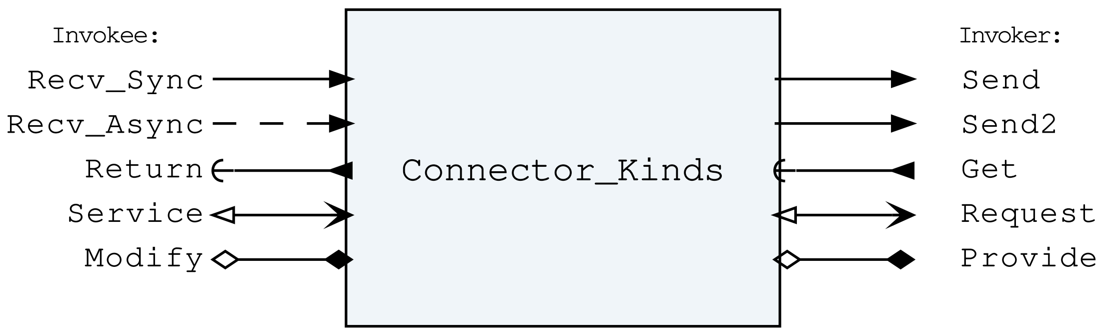

Figure 1: Example component diagram to demonstrate different connector *kinds*

The different arrow types follow a few principles. The dark, filled in, portion of the arrow is always connected to the *invokee* component, the component responsible for performing the behavior of that connector invocation. The *invoker* side of the connector has a more “open" representation. The direction of the arrow in each connector representation signifies the direction of the data flow. For instance, *get-return* connectors use backwards facing arrows to signify that the data flow direction is opposite of the invocation direction, and *provide-modify* connectors use diamonds, instead of arrows, because data is flowing in both directions via a single modifiable type. Note that `recv_async` has a dotted line instead of a solid line like the `recv_sync` connector, to demonstrate its asynchroneity. A `send` connector will always have a solid line representation in a component diagram since the *invokee* component is responsible for declaring a compatible connector as either `recv_sync` or `recv_async`.

Note that connectors have a many-to-one relationship. This means that for any *invokee* connector, many *invoker* connectors can connect to it. For example, if a component has a single *invokee* connector which takes a data item and logs it to disk, many other components can connect and send data items to this single connector. In order to create a one-to-many relationship an arrayed connector can be used, which is a set of connectors of the same type accessible by index. This design might be useful for a command router component, which takes in a single command but must route it to the correct destination component, which is connected to a specific index in the arrayed *invoker* connector. Examples of both types of connections are explained in the `Components` section of this document.

## 3.2 Packed Records

In general, any datatype can be sent along a connector from one component to another. However, if two components live on different processing units, differences in data representation and endianness can cause compatibility issues. The Adamant framework provides a mechanism for dealing with this called *packed records.*

A packed record is a structure-like data type which can contain an arbitrary number of fields of any type, including arrays or other packed records. Packed records have two representations, a native machine representation, and a packed representation. The packed representation can be thought of as the “serialized" version of the record. The storage size used for fields in the packed representation can be tailored by a user of the framework. The packed representation is always stored in big endian byte order. A conversion between the packed and unpacked versions of the records is also provided.

Using packed records for component connector types is preferred in order to increase the compatibility of components across distributed systems.

## 3.3 Components

At the core of the Adamant framework is the concept of a *component.* A component is a module of code that encapsulates behavior behind a well defined interface. A component's interface consists of a set of connectors. The use of connectors makes each component independent from other components.

In other words, components do not depend on one another, they only depend on their own connectors. This decoupling of components is the primary mechanism by which the Adamant framework aims to improve the reusability, testability, and adaptability of embedded software.

The interworking of components are described in the next few subsections.

### 3.3.1 Component Queue

In Adamant, a component is defined by declaring its interface in a component model, which consists of a list of connectors. If any of these connectors are of kind `recv_async` then the component is constructed with an internal queue to store messages for processing at a later time.

In some situations, asynchronous communication might be advantageous. Consider a component whose task is to log data to disk. In order to allow other components to send data to this logger component without waiting for the entire disk write operation to complete (which might be very slow), an asynchronous connection might be used in order to queue the data in memory. In this way, messages sent from components to the logger component will be immediately queued up, allowing those components to continue their normal operation without waiting for the disk write to complete. When the logger executes at some later time (most likely at a lower priority), it will be able to store these messages to disk.

Components in Adamant are constructed with at most one queue to service all asynchronous messages. This design ensures that the component can only wait on a single message stream, simplifying the ability to reason about its behavior.

Adamant provides two different queue types which can be chosen by a component designer to best fit their use case. The standard Adamant queue is efficient in both space and time, allowing variable sized messages to be stored without any wasted bytes on the queue. Items are dequeued in the same order that they are enqueued (first-in-first-out). Adamant also offers a priority queue for when it is critical to dequeue certain items before others.

### 3.3.2 Component Execution

Components in Adamant are not necessarily associated with a task. A component can be designed to execute only when its connectors are invoked, or it can be given a dedicated task of execution, or it can be constructed to function in both cases. The different component *execution* types are described below:

- **Passive** - `Passive` components do not have a thread of their own and are expected to execute on the thread of their connector invokers.
- **Active** - `Active` components contain their own thread on which they execute.
- **Either** - `Either` components are designed to function either as `passive` or `active` components. Which *execution* type a component will get resolved to is decided at the assembly level.

In this paradigm, not all components have their own thread of execution. This makes the component model and the thread model in Adamant distinct and separate, providing flexibility in the way components can be put together and executed in different systems.

The combination of component queues and component execution creates a few distinct categories of components, all which excel at accomplishing different tasks. Below is a discussion of each pattern.

- **Passive without Queue** - This type of component will only execute when a caller invokes one of its connectors. This type of component is typically used to provide a synchronous service to other components, such as getting the system time, or updating a parameter database.
- **Passive with Queue** - Like the *Passive without Queue* component, this component will also only execute when called. However, because it has a queue, this component is typically used to respond to asynchronous messages at a scheduled time. Execution is commonly invoked through a “schedule" connector, which tells the component to service its queue and possibly do

other periodic work. This design is ideal for components that need to be executed with cyclic real-time deadlines, such as a control loop.

- **Active with Queue** - The default behavior of components that follow this pattern is to block on their queue and only wake up and do work when a message is received asynchronously. This type of component is ideal for components that run in the background and receive asynchronous messages, such as a logger. This default task behavior can be overridden if necessary by the developer, but this should be avoided if possible, since the component diagram will not give insight into this custom behavior.
- **Active without Queue** - Unlike *Active with Queue* components, there is no default behavior for the *Active without Queue* task. In this case, the developer must define the code which executes on the component's thread. This component pattern is uncommon, and not recommended, since the component's behavior cannot be readily ascertained from its diagram. However, this design might be useful for components that run in the background, but receive no asynchronous input, such as a memory scrubber.

Further discussion on component *execution* is provided in the section *Assemblies*.

### 3.3.3 Component Examples

The truly defining aspect of a component is its interface of connectors. Below is a set of example components which exhibit some common connector patterns useful in embedded software systems. These diagrams contain a box, representing the component, which includes the *type* of the component. Each connector is labeled with its name and the appropriate symbol representing its connector kind. See the *Connectors* section for details on connector kind symbols. Connectors shown on the left side of each diagram are called *invokee* connectors, meaning the behavior of the connection invocation is implemented within the component itself. Connectors on the right side of each diagram are called *invoker* connectors, meaning they initiate the invocation of behaviors within the outside components to which they connect.

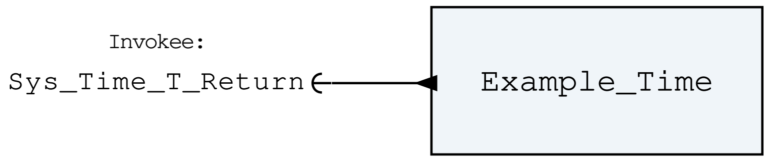

Figure 2: `Example_Time` component which returns the current time when asked

This `Example_Time` component has a single *return* connector, which returns the current time upon request. Note that because of the many-to-one relationship for connectors, many different components can connect to the `Time_Return` connector to obtain time. The time component does all its work synchronously so does not need an internal queue.

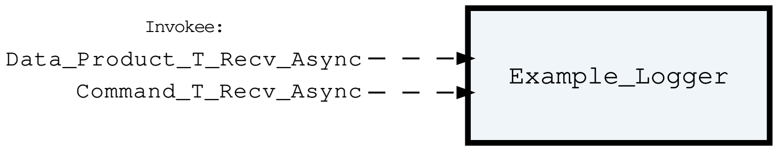

Figure 3: `Example_Logger` component which logs telemetry to disk

This `Example_Logger` component has two asynchronous *invokee* connectors. For this reason, it is created with an internal queue. The purpose of this component is to log telemetry items to disk. Telemetry items come in asynchronously via the `Data_Product_T_Recv_Async` connector, and are logged on demand. This component also includes an asynchronous command *invokee* connector for receiving configuration commands. A component like this would most likely be associated with a low priority task that runs in the background.

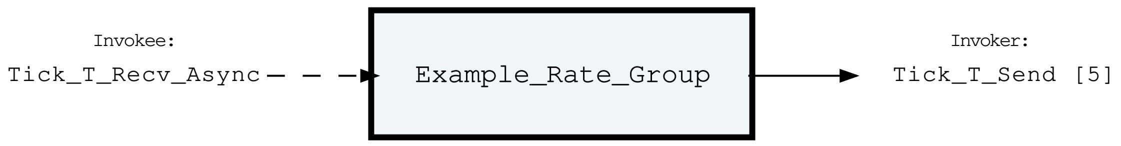

Figure 4: `Example_Rate_Group` component which takes a tick in (maybe from a hardware interrupt) and schedules the execution a set of components connected to its *invoker* connectors

The `Example_Rate_Group` is useful for scheduling the activity of other components in a system. It takes a single synchronous connector of type `Tick.T` in at one rate and schedules the execution of components connected to its arrayed `Tick_T_Send` connectors using its internal task. This component could be used to take a 1 Hz hardware driven interrupt and schedule the execution of components throughout the system that need to execute at a 1 Hz cadence.

Note that each index in a connector array acts the same as a normal connector. Each index could be connected to connectors on disparate components, or to different connectors on the same component, or even to the same connector on the same component.

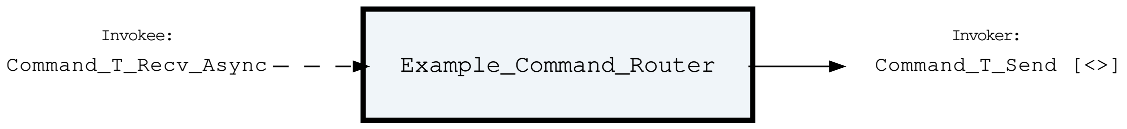

Figure 5: `Example_Command_Router` component which takes in commands and sends them to the appropriate destination component

This `Example_Command_Router` component's job is to take commands asynchronously from an *invokee* connector, determine where that command should be routed to for execution, and then send that command out the correct *invoker* connector. Note that the `Command_T_Send` connector has an array length of <>. This denotes that the size of the array will be determined during the construction of the assembly. The component itself is made more generic by not having to hard code this connector array length into the component model.

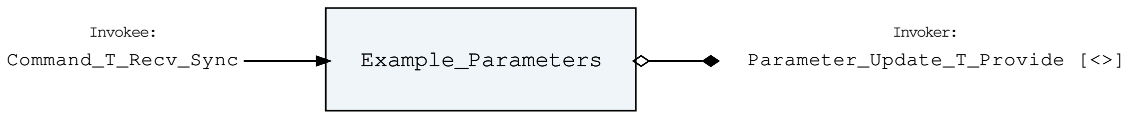

Figure 6: `Example_Parameters` component which takes requests for parameter values by identifier, and synchronously returns them to the requester

The `Example_Parameters` component's primary task is to push the current value of parameters to components throughout the system. To accomplish this, it has an arrayed *provide* connector through which it sends parameter updates to destination components. The component also includes a connector which allows parameter values to be changed by command.

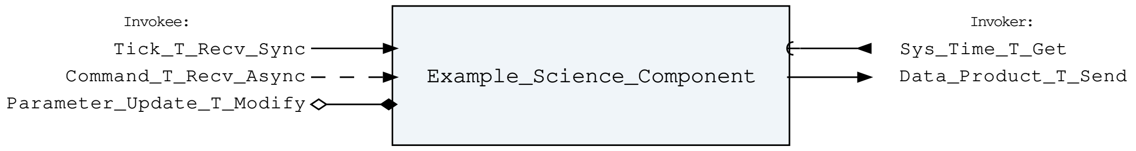

Figure 7: `Example_Science` component which is scheduled by a tick, produces telemetry, and can be configured by command

The `Example_Science` component is more complicated than the rest, since this component might perform the actual function of the mission. In this case, the component's execution is scheduled via a `Tick_T_Recv_Sync` connector. The component can be configured asynchronously via command, and is expected to produce telemetry. It receives parameter updates through its `Parameter_T_Service` connector. To time stamp the telemetry, this component also has a connection to get time.

In the *Assemblies* section, we will see how all these components can be combined together to form a functioning executable network.

## 3.4 Tester Components

A vital part in proving that a component is ready for deployment and meets its design requirements is unit testing. Because component interfaces are a well defined set of connectors, the testing interface is known. Adamant automatically assists users by generating a tester component from the component model. The resulting tester component contains connectors of the same *type* as the original component but of opposite *kind*. For instance, if a component has a `get` connector with *return_ type* `T`, its tester component would be generated with a `return` component with *return_ type* `T`. In this way, the tester component can simulate the component's system context during test, and can be used to drive the component through specific testing scenarios.

As an example, the generated tester component for the `Example_Science` component, discussed in the *Components* section, would look like:

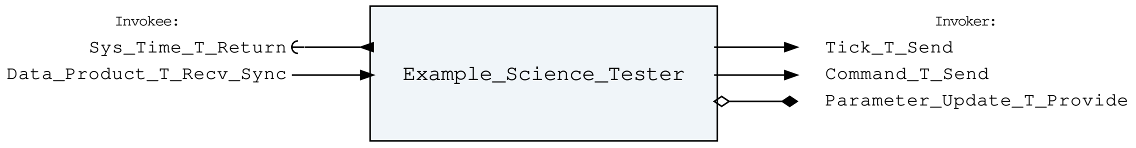

Figure 8: The `Example_Science_Tester` component, which has a reciprocal set of connectors to the `Example_Science` component

Notice that for every *invoker* connector in the `Example_Science` component, there is a compatible *invokee* connector in the `Example_Science_Tester` component. The analogous is true for the *invokee* connectors on the `Example_Science` component. Adamant enforces that all tester components be generated with only synchronous *invokee* connectors, so as to not need an internal queue. This eases the orchestration of the test component by the test suite, which makes writing tests easier.

When unit testing commences, the tester component can be asked by the test suite to connect to the component under test. This is shown in an assembly diagram below. Component assemblies in Adamant are discussed in the following section.

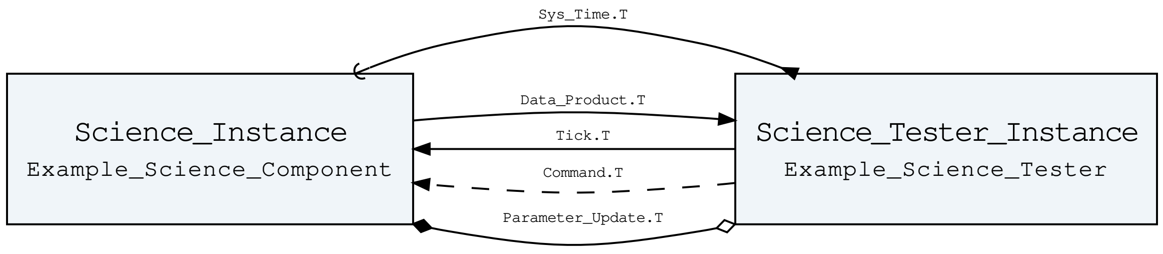

Figure 9: An assembly diagram showing how the tester component is connected to the component under test

It is important to understand that the tester component itself does not contain unit tests. Rather, it provides the necessary interface to communicate with a component and simulate its execution context. To actually test the component, a developer would create a unit test suite which instantiates a tester component during execution in order to exercise functionality within the component under test. When a tester component is instantiated, it provides not only methods for exercising the component through connectors, it also exposes the component internals, allowing for full “white-box" testing. This gives the user the ability to verify component execution at a much finer granularity than invoking connectors alone. To speed up test development, Adamant also provides facilities to assist in the generation of scaffolding code for unit test suites, which is discussed in detail in the *User Guide*.

## 3.5 Assemblies

After components have been modeled, implemented, and tested, they are ready to be integrated into an *assembly*. An assembly model identifies the components to be included in a software deployment and the specific connections between them. An integrated assembly of the example components presented in the *Components* section is shown below.

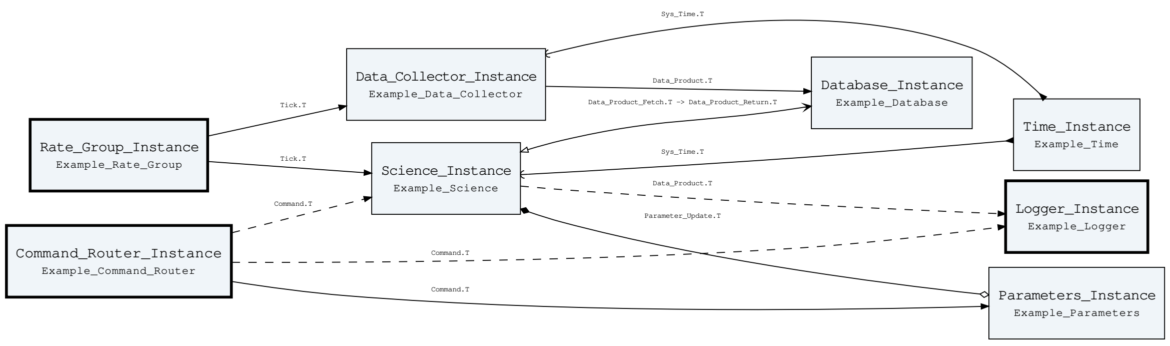

Figure 10: An assembly diagram showing how components can be connected together

Components in the assembly are labeled with their *name* first, followed by their component *type* on the subsequent line. In this assembly, we can see that the `Rate_Group_Instance` component provides a tick to drive the `Science_Instance` component's execution. Commands are sent from the router to the `Parameters_Instance`, `Logger_Instance`, and `Science_Instance` components. The `Science_Instance` component also has visible interactions with the `Parameters_Instance`, `Time_Instance`, and `Logger_Instance` components. Each connection maintains its symbol in order to denote the connector *kind* and is labeled with its *type* and/or *return_ type*.

Note that some components in the assembly diagram are outlined in bold, while others are not. The boldness of the outline designates the component's *execution* property as either `passive` or `active`. `Active` components are given a task in which to execute and are bolded in assembly diagrams. Passive components are not given a task, and are expected to be executed on the thread of calling components. In this assembly, there are 6 components, but only three tasks used to orchestrate their execution.

Whether a component is `active` or `passive` greatly effects its behavior within an assembly. For instance, component with an internal queue is also `active`, by default it will block on its queue, waiting for work. When messages arrive asynchronously, it will execute on those messages when it is given time to run by the runtime or operating system scheduler. This design decision is usually good for tasks that are not real-time critical, and need to get work done on-demand in the background. The `Logger_Instance` component, shown in the assembly above, demonstrates this pattern. Conversely, if a component with a queue is `passive` and not given a task in which to execute, it must do work on the thread of a calling component. During this execution, the component might process a maximum number of messages from its queue. This is demonstrated by the `Science_Instance` component, which is called periodically to do work by the `Rate_Group_Instance` component. In this scenario, the work done by the `Science_Instance` component is actually executed on the thread provided by the `Rate_Group_Instance` component. When called, the `Science_Instance` component would perform a set amount of work, responding to commands in its queue and emitting telemetry from its instrument. This second method of scheduling is better for components that need to be executed in a real-time manner, as their execution times can be synced with other timing events throughout the system, and monitored for deadline overruns.

Besides using assemblies to model the architecture of the entire embedded system, assemblies can also be tailored to address certain needs during testing. For instance, smaller, more focused assemblies can be used during development to answer specific questions, provide insight into trade studies, or to isolate testing to a single subsystem while not worrying about the complexity of the rest of the system. In this way, assemblies are adaptable to the changing needs of a project through its life cycle.

Moreover, because the interfaces of components are very well defined, swapping real components for simulation components in an assembly can easily be achieved. A common use case might be swapping out the real deployment version of a complex component for a simpler one during initial development. For instance a complicated communication component which talks over a 1553 bus could be swapped with a simpler component that communicates over TCP/IP sockets during testing on a development computer or in the absence of communication hardware. Another example might be replacing a small network of components, say a set of components which interact with a science instrument, with a single component that simulates that science instrument in order to validate the remainder of the system. Finally, different components which perform the same task can be designed specifically for execution on different architectures. The choice of which component to use for a particular test or production deployment depends on the target hardware. The rest of the system can remain unchanged. The swappable nature of components supplies an endless myriad of possible solutions when it comes to developing reliable software in the face of unknown or unavailable hardware, differing interfaces, and evolving requirements.

## 3.6 Views

The assembly diagram shown in the previous section is admittedly complex, even for the very simple network of components presented. In order to provide a more digestible way of consuming the information in an assembly model, the concept of assembly *views* is introduced. A view is a diagram that only shows a subset of the entire assembly model in order to demonstrate a particular point to a set of stakeholders. Because a view is focused, and not cluttered by information that may distract from its main message, it is usually more adept at presenting certain aspects of the architecture. A collection of views can generally communicate architectural ideas much more effectively than an entire assembly diagram, making for more relevant discussions and more valuable design reviews.

For example, a view of the previous assembly model can be created that shows only the command paths of the architecture.

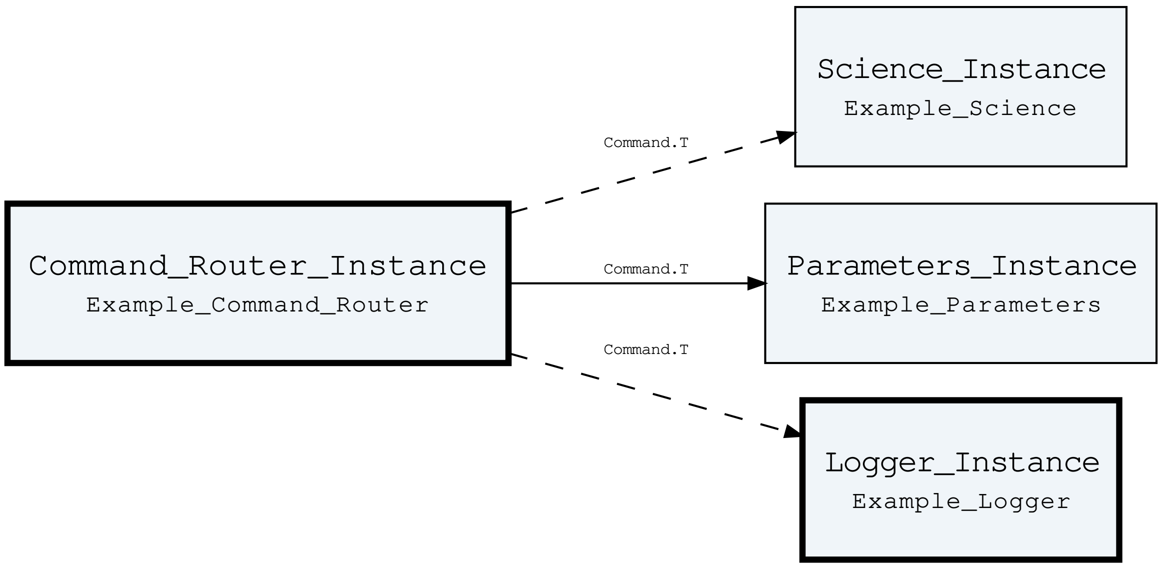

Figure 11: A view diagram isolating the command path of the assembly

This view clearly shows that there are three command paths, all originating from the `Command_Router_Instance` and ending in three components responsible for executing commands.

Another example of a view might be one that shows the assembly from the context of a single component. For instance, below is a view that shows the `Parameters_Instance` component and every other component that interacts with it directly.

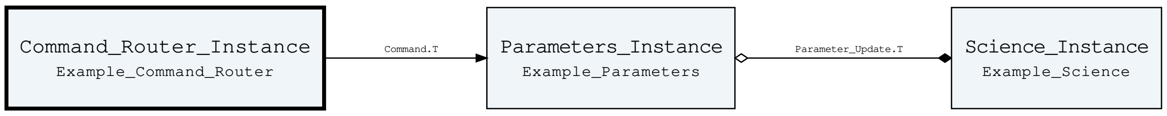

Figure 12: A view diagram showing all connections made to the `Parameters_Instance` component

Views can be created to show any subset of information available in the assembly. They are generated by creating a view model, in which a designer specifies which architectural entities they are interested in showing or which entities they wish to exclude.

## 4 Conclusion

The Adamant framework provides many facilities to aid in the development of real-time, embedded systems. Its component-based architecture allows for the encapsulation of behaviors within independent modules. Components can be designed such that they are reusable between projects, and are easily swappable for compatible components when reuse is not possible. All of the component descriptions and the connections between them are documented in textual models which can readily generate diagrams of the system. This visual representation of the architecture is guaranteed to be directly reflected in the actual software implementation through the use of code generation from the same textual models. For these reasons, the component-based, model-backed architecture provided by the Adamant framework provides many advantages over standard embedded software development practices.