Introduction

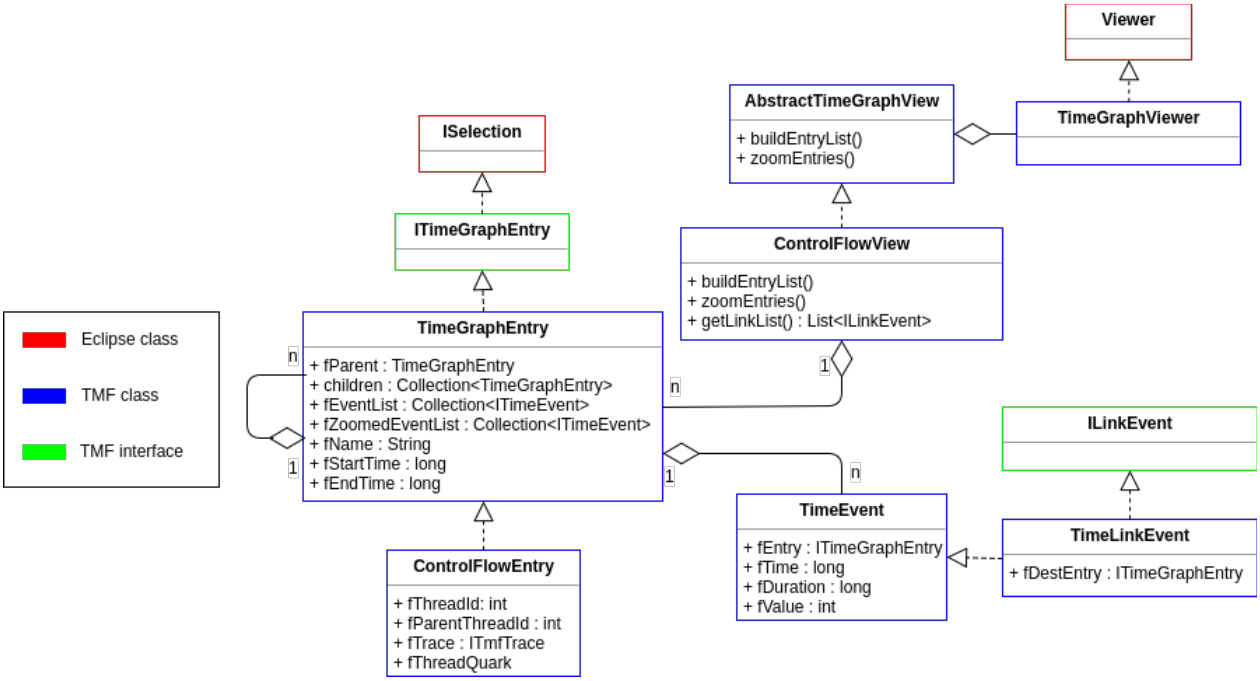
The purpose of this document is to suggest a new architecture for the time graph views by introducing a layer of abstraction. This architecture would decouple the UI from the data model and the core, which increases the maintainability, testability and portability of the code base.

To decouple UI and core, we want to extract a serializable model that can populate other views, for instance, a JavaScript one. This was already done in the XY Line/Scatter charts. A data provider pattern shall be implemented for time graph.

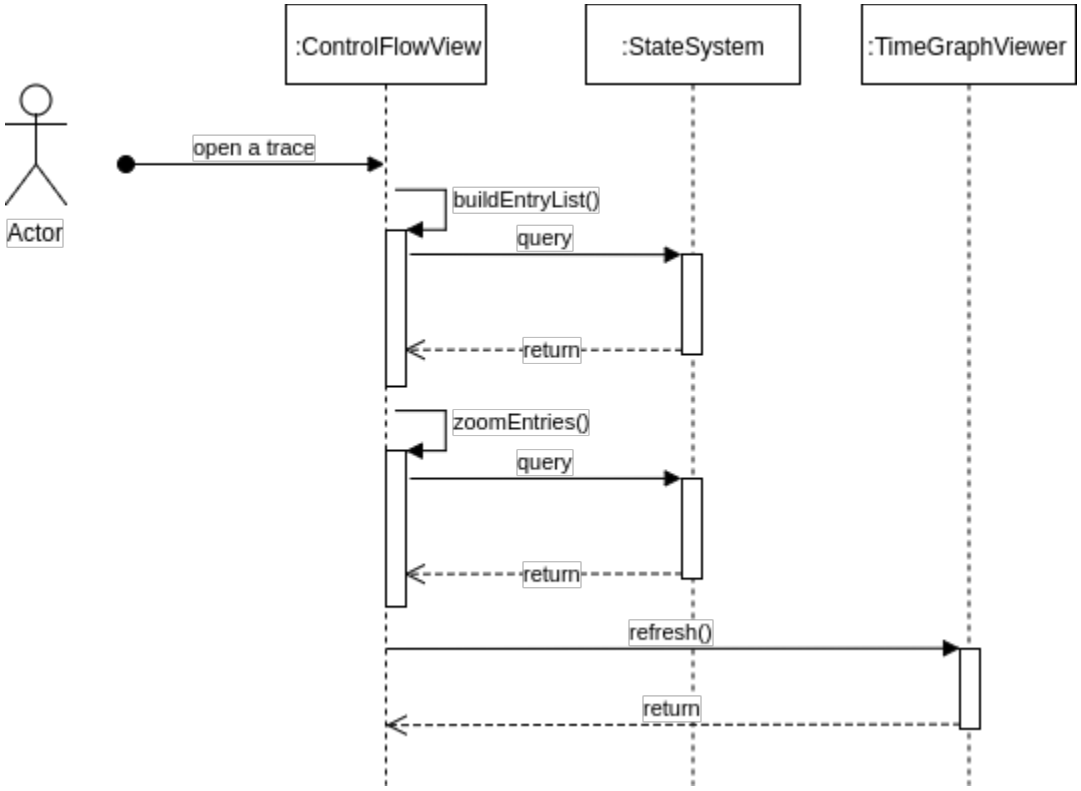
A high-level view of how time graph is currently working follows.

Current architecture

This is a very simplified class diagram of the current architecture of the control flow view.

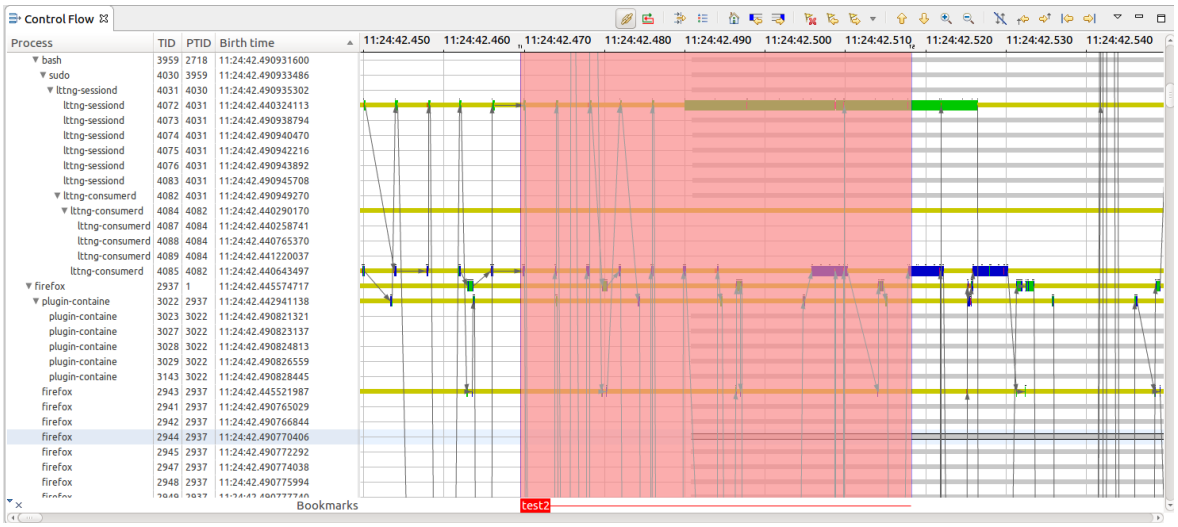


Other views that inherit from AbstractTimeGraphView work relatively the same way as the control flow view. Thus, following sections will use control flow view as example. Let’s understand how those classes interact between each other by showing a simple use case: user opens a trace.



1. User opens a trace.
2. Important methods called in **ControlFlowView** that build the current model are:
    a. buildEntryList() uses a State system to build a collection of **ControlFlowEntry.** This method only builds the tree on the left side of the view. **ControlFlowEntry** is a composite pattern.
    b. zoomEntries() uses a State system to replace the event lists for each **ControlFlowEntry** previously built by buildEntryList method. A collection of **TimeEvent** is created for each **ControlFlowEntry.** A **TimeEvent** is a rectangle of color on the right side of the view which represents a state between two timestamps.
3. Once the **ControlFlowView has** finished building its model, it calls the **refresh** method that notifies the **TimeGraphViewer** that the model is ready. The viewer then draws the model.

    Graphically, a **ControlFlowEntry** is a row in the control flow view. It encapsulates a list of **TimeEvent** shown as colored rectangles. In addition, the view also has arrows (implemented as a collection of **TimeLinkEvent**), bookmarks (shown in red selection in the figure) and tool tips on **TimeEvent** and on **TimeLinkEvent**.



The main issue of this architecture is that **ControlFlowView** class is responsible for computing the model. Thus, it is highly coupled to the Eclipse platform and we want to limit that.

The following section presents the data provider pattern applied to time graph. This document does not cover the solution for bookmark and tool tip. It focuses mainly on time graph rows (entry on left and list of time events) and arrows. A data provider pattern could be applied to bookmarks and tool tips too.

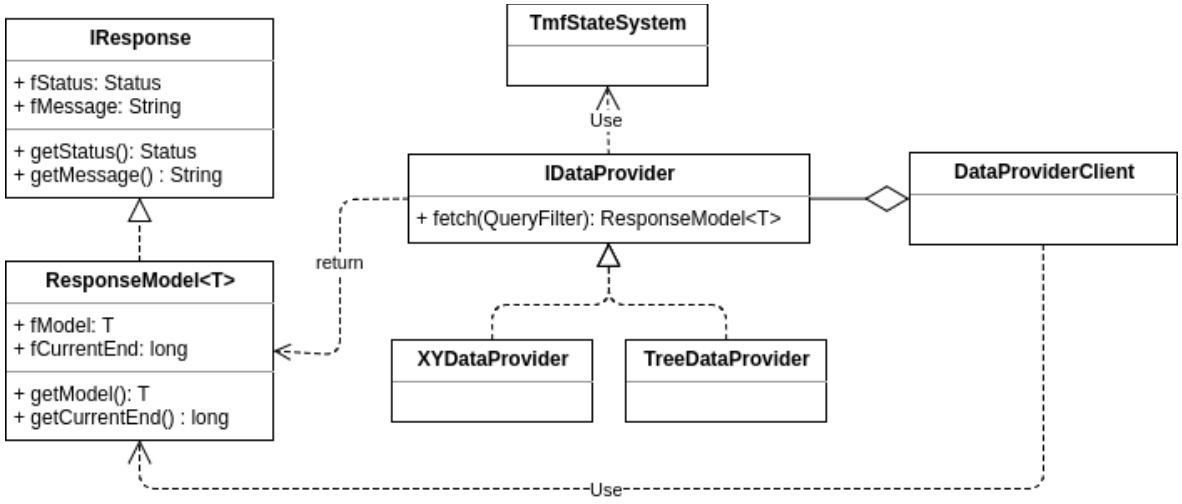
Data Provider pattern

The Data Provider pattern is greatly inspired by the Repository/Service pattern.

- The Repository layer is an abstraction on top of the data access. Its responsibility is to retrieve data, regardless of how it is stored (SQL database, MongoDB, simple file). In our case, we can consider the state system as the Repository layer.
- The Service layer is built on top of the Repository layer. This layer contains business logic. Indeed, once the data is retrieved, the classes of this layer are responsible to manipulate/modify it. In our case, a Data Provider could be considered as a Service.

The Data Providers build and return a serializable model. The created model should be simple (least external dependencies), ready to render and immutable. In addition, the Data Providers should be RESTful as much as possible.

As HTTP responses, it will also work with status. The following class diagram shows how Data Providers are used for XY Charts and TreeViewer.



- The Data Provider's clients only manipulate a **ResponseModel**. They don't care how it is built. They only know an **IDataProvider** which is responsible to return a **ResponseModel**.
- The **ResponseModel** encapsulates a **Status**. In our case, a status could be: RUNNING, COMPLETED, CANCELED or FAILED. Then, it's the data provider client's responsibility to deal with each status (requesting again, log error, stop requesting, etc.)
- The **IDataProvider** needs a QueryFilter. Basically, a QueryFilter encapsulates all required information to compute a model. For example, the CpuUsageDataProvider (a subclass of XYDataProvider) needs the selected thread and CPU to compute the CPU usage for the given thread or CPU.
- The classes implementing **IDataProvider** are responsible for building the specific model. For example, **XYDataProvider** will always create an **XYModel**.

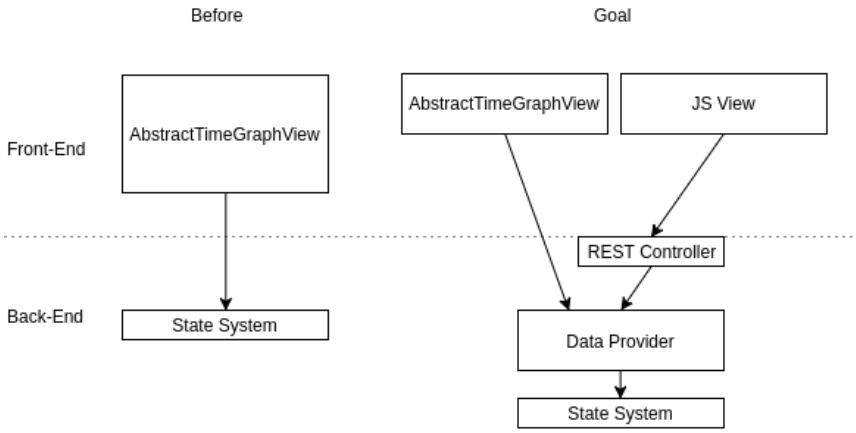
Suggested architecture

The design goals of the suggested architecture are:

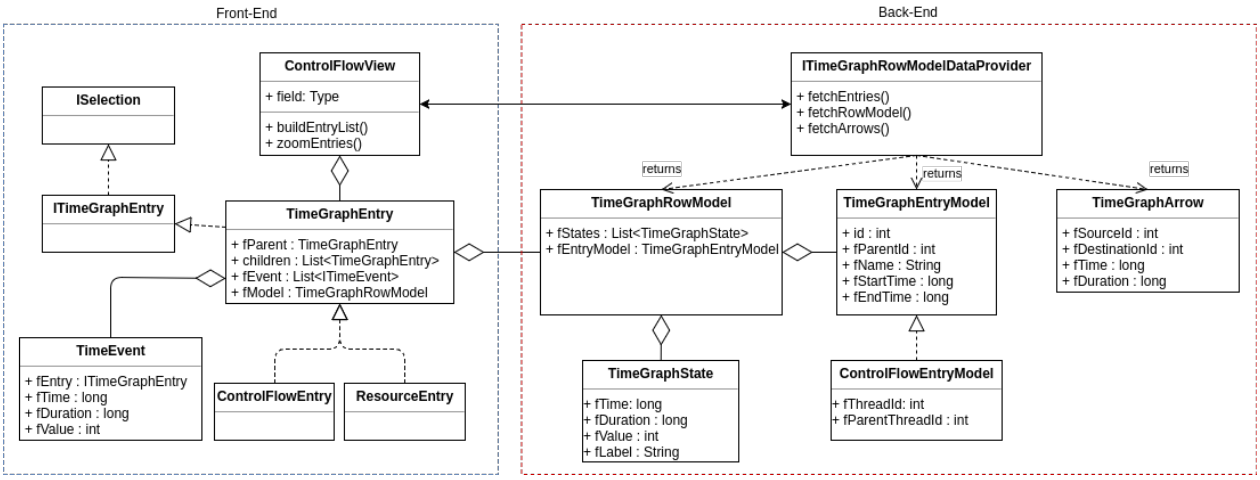
- A "ready to render" model. Except for getters, no methods should be available. The model should be immutable and serializable.
  - Serializable models are more testable and support multi-language clients.
  - Immutable models are inherently thread safe. This will also lead to more stable builds and more accurate views.
- In order to reuse the model with other micro services such as a JS view, we need to limit the size of the model, to minimize the volume of exchanged data, (especially in the case of remote servers). Querying only the required data is a possible approach to limit the size of the model.

- A new layer of abstraction between the view and the data, **Data Providers.** Classes of this layer are responsible for computing and returning the model.
- To limit API break and preserve backward compatibility when possible.
- RESTful

The TimeGraphEntry and **ControlFlowEntry** classes and interfaces shall be kept. They will be built by composition in the new model instead of inheritance (to avoid a diamond pattern). The logic behind computing the model is moved to the **Data Provider** layer. This layer will be in core packages.



Classes presented in the previous class diagram should be preserved. They are part of “Front-end” layer. Now, let’s introduce new classes in order to reach the goal architecture.



The blue dotted rectangle preserves the current architecture and **isolates the current classes in the front-end**. Instead of keeping fields like name, start time, end time, the **TimeGraphEntry** now encapsulates a **TimeGraphRowModel** to avoid breaking the API. Classes inside the red dotted rectangle are part of the core package and should be common/reused/extended by other TimeGraphView

implementations. To support grouping, the **TimeGraphEntryModel** has his parent's ID. It's the view's responsibility to show either a flat or hierarchical structure.

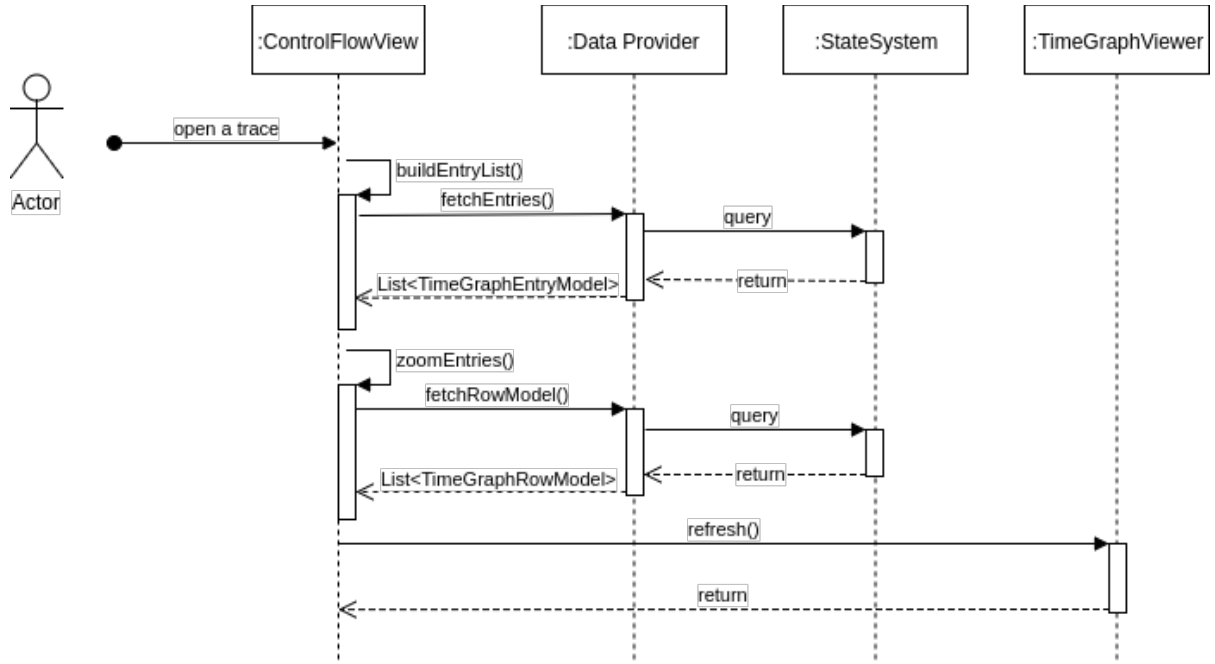
Before/After analogies

- **TimeGraphRowModel** is a model for the current **TimeGraphEntry**. However, it no longer references its parent directly but has a parent ID (int) field. We use the parent's ID to rebuild the tree in the front end, thus avoiding double linkage in the model and slightly reducing its size.
- **TimeGraphState** is a model for the current **TimeEvent**. For the **ControlFlowView**, some states must show a label, for example system calls. So, when needed, **TimeGraphState**'s label field should be set only if the size of the rectangle allows to. An other stategy to limit the model size is to use a presentation provider for repeating labels.
- **TimeGraphArrow** is a model for the current **TimeLinkEvent**. It no longer references its source and destination entries directly but has a source and destination ID.

All models in red dotted rectangles are immutable. They are intended to be simple and contain only "ready to render" information.

Use case: user open a trace

Now, let's see how Data provider impacts the use case presented in previous architecture.



With the **Data Provider**, the **ControlFlowView** class is no longer responsible for computing and building the model. Its logic is moved to fetch methods of the data provider and returns a “ready to render” model. The model contains only data, there is no information about the UI (Color, thickness, etc). The view's responsibility is to query the data provider with a start time and end time. Information about UI (Color, thickness) will be handled by a presentation provider.

ControlFlowView is still responsible for querying the Data Provider until the analysis is complete. Indeed, **ControlFlowView** must handle the ModelResponse according to its Status. If Status is RUNNING, for example, the computed model is partial and **ControlFlowView** should wait X milliseconds before requesting the Data Provider again.

Data provider: main methods

Since we want to limit data transfer, here is a description of available methods of the data provider, their usage, the parameters and what they return.

Method	Usage	Parameter	Returns
fetchEntries	Used only for retrieving data about the left side of the control flow view.		A list of **TimeGraphEntryModel**
fetchRowModel	Used only for building the events of the time graph.	Start time End time Resolution List of TimeGraphEntryModel ids	A list of **TimeGraphRowModel**
fetchArrows	Use only for retrieving arrows.	Start time End time Resolution	A list of **TimeGraphArrows**

Estimated sizes of transactions and amount of transactions

- **fetchEntries**() : the size of the model is proportionate to the number of entries in the trace. This method should be polled until the analysis is complete, it does not change afterward.
- **fetchRowModel()** : the size of the model is proportionate to screen size (because we have a virtual view that queries only visible items). Worst case is having one event per horizontal pixel and one entry per vertical pixel. For a FullHD screen, that means ~2 million elements. This method is called every time a zoom/pan is done on the time graph.

- **fetchArrow()** : for the **ControlFlowView**, the number of arrows is proportionate to the number of horizontal pixel and the number of CPUs. This method should be called every time a zoom/move action is done on the time graph.

Comparison of two architectures:

Potential gains:

- Testable
- Maintainable
- Reusable
- Reduced responsibility of the view

Potential issues:

- Performance due to copying the model into the ViewModel
- Complexity
- Response time
- Presentation provider and tool tip handling? Making sure all the features still work.