

From Use Cases to Code

(A Case Study)

Simon Patton
(LBNL)

Caveat

- **Process is not linear, really a random walk.**
 - Include elements of the following:
 - the initial Use Cases;
 - previous experiences;
 - impact of development tools;
 - assumptions that need to be made to handle issues not addressed in Use Cases.
- **Process is iterative**
 - This is only **first** iteration.

First Steps

- **Define scope of iteration.**
 - Provide a **basic** implementation of the Frame Analysis Executable (Faye)
- **Select appropriate Use Cases.**
 - 8.1 Analyze Frames in a Data Set.
 - 8.2 Configure the Frame Analysis Executable.
 - 8.3 Analyze a Frame in a Data Set.

8.1 Analyze Frames in a Data Set

Use Case 8.1 Analyze Frames in a Data Set

Description: When a physicist wishes to analyze data from the detector, this data is presented in Frame. Each new Frame that is presented it can be processed by one or more algorithms to derive new data or make decision based upon existing data.

Precondition: The Frame analysis executable exists.

Postcondition:

Other Requirements:

Primary Scenario:

- 1) Create a new module which contains one or more algorithms which can be applied to an active Frame.
- 2) Start the Frame analysis executable.
- 3) Configure the executable (Using Use Case 8.2 Configure the Frame Analysis Executable).
- 4) Analysis a specified number of Frames (Using Use Case 8.3 Analyze a Frame in a Data Set).
- 5) Inspect the current state of the jobs output, e.g. histograms, etc.
- 6) Decide on changes to configuration, e.g. modified module parameters.
- 7) Rewind input to the start of the data set.
- 8) Repeat steps 3) to 7) as necessary.
- 9) Terminate the Frame analysis executable.

8.2 Configure the Frame Analysis Executable

Use Case 8.2 Configure the Frame Analysis Executable

Description: The Frame analysis executable is a flexible framework in which an number of different analysis can be performed. The exact analysis which is performed is dependent of how the executable is configured.

Precondition: An instance of the Frame analysis executable is running.

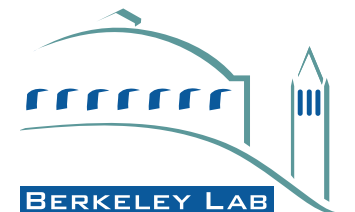
Postcondition: The executable is ready to analyze Frames.

Other Requirements:

Primary Scenario:

The following activities can happen in any (sensible) order:

- 1) Create an instance of an input module.
- 2) Bind input data sets to any input modules which can read the appropriate formats.
- 3) Create an instance of an algorithm.
- 4) Bind algorithm to one or more appropriate Streams or to an appropriate proceeding operation.
- 5) Create an instance of an output module.
- 6) Bind output data sets to any output modules which can write the appropriate formats.
- 7) Configure options on any module, e.g. change modules parameters.



8.3 Analyze a Frame in a Data Set

Use Case 8.3 Analyze a Frame in a Data Set

Description: The Frame analysis executable can analysis Frame without any user interaction (after it has been configured and told to analysis the Frame). However the user needs to have an expectation of how the execution of the analysis proceeds in order to be able to usefully configure the executable.

Precondition: An instance of the Frame analysis executable is running, has been configured and has been signalled to read on or more Frames.

Postcondition: The Frame analysis executable is in a state which allows it, or any module or algorithm to be reconfigured.

Other Requirements:

Primary Scenario:

- 1) Find the next SyncValue available from any Active Stream
- 2) Create a Frame with this SyncValue, using all configured input modules.
- 3) Execute all algorithms which are bound to the Active Stream which caused the Frame to be built, and any succeeding algorithms.
- 4) Execute all output modules, which output any portions of the Frame that they are configured to store.
- 5) Dispose of the Frame.

Find Classes and Interfaces (1/2)

- **PlugIn**: A common interface for all algorithms (UC 8.1[1] and UC 8.2[3]).
- **Faye**: The self-contained executable (UC 8.1[2] and [3]).
- **Source**: A common interface for all input modules (UC 8.2[1]).
- **Stream**: Represents the Stream associated with a Record (UC 8.2[4]).

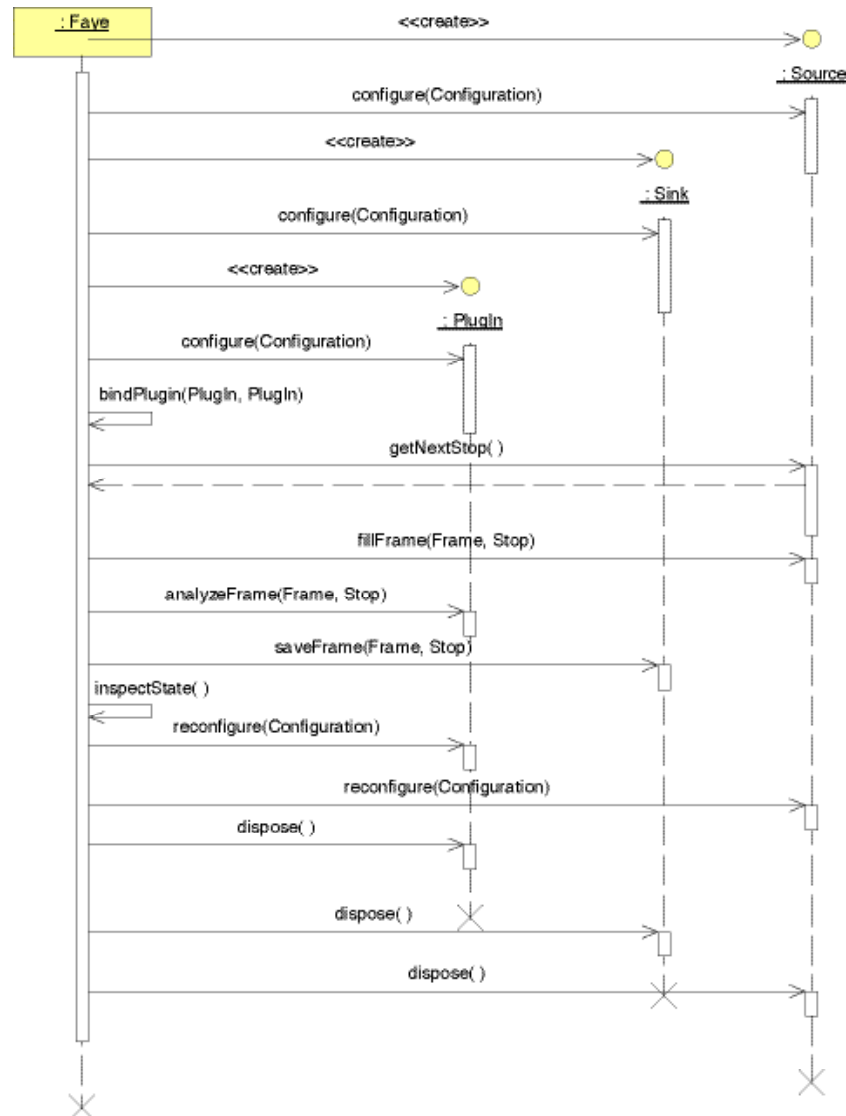
Find Classes and Interfaces (2/2)

- **Sink**: A common interface for all output modules (UC 8.2[5]).
- **Parameter**: A value that can be set in a module (UC 8.2[7]).
- **Frame**: A placeholder for the Frame, whose details are not required in this iteration (UC 8.1[4] and UC 8.3[2]).
- **SyncValue**: Represents the ordering of Record (UC 8.3[1]).

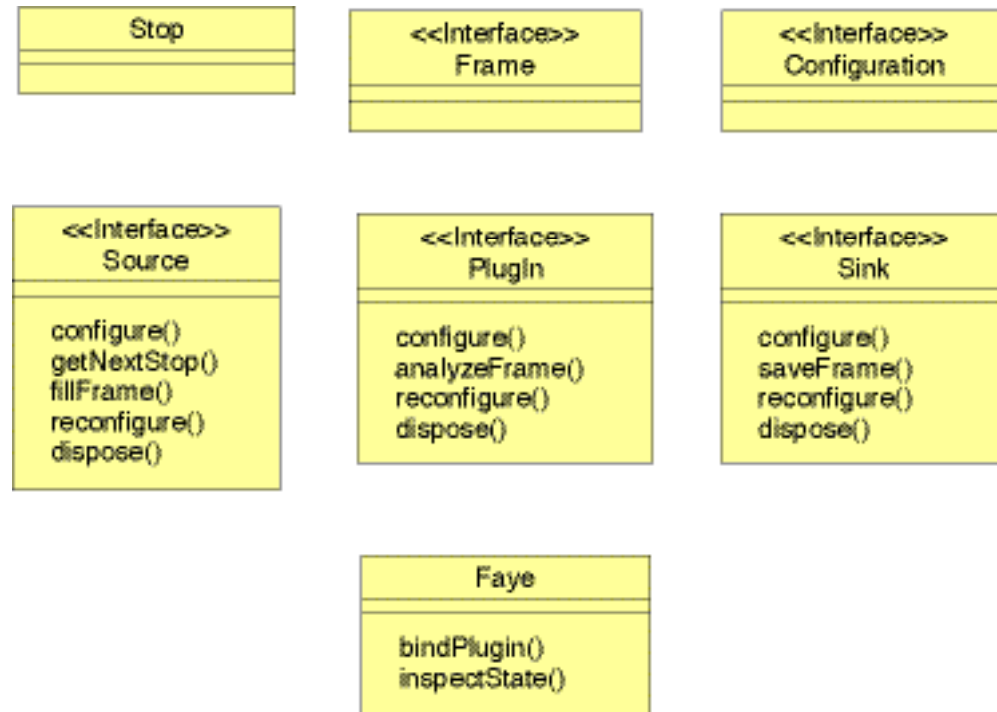
Find and Characterize Collaborations

- **Alan Kay, the developer of Smalltalk:**
 - “A program is a bunch of objects telling each other what to do by sending messages.”
- **Document interaction between objects use:**
 - interaction diagrams
 - collaboration diagrams
 - state diagrams

Interaction Diagram for Faye



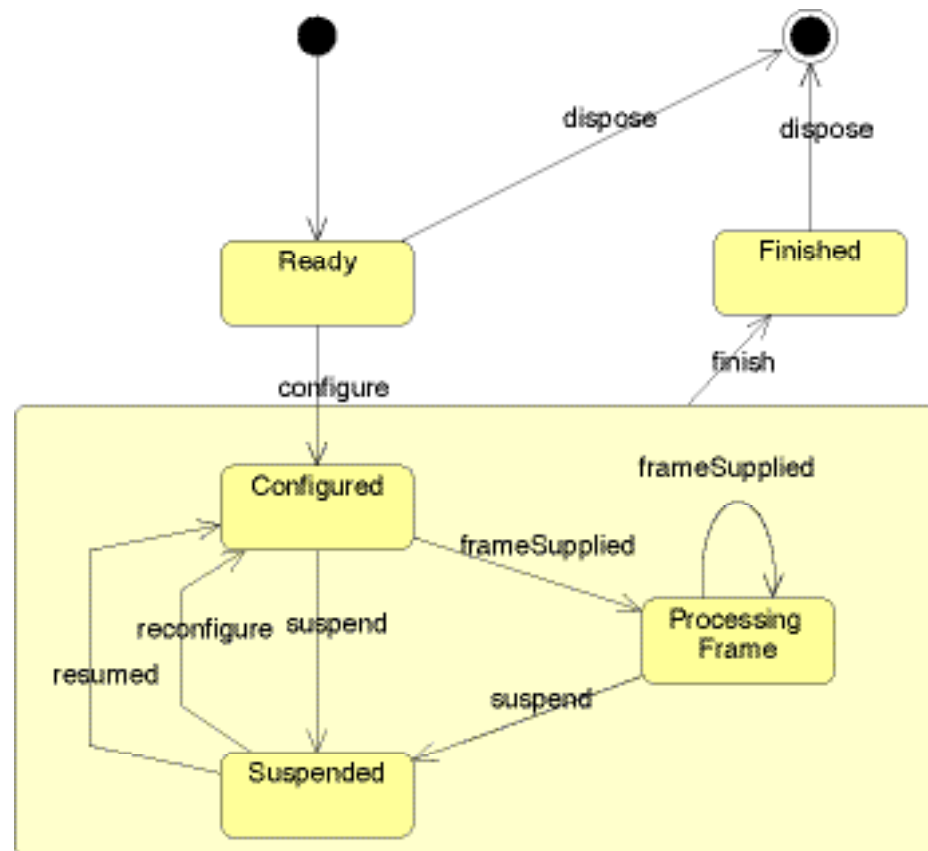
Derived a Class Diagram for Faye



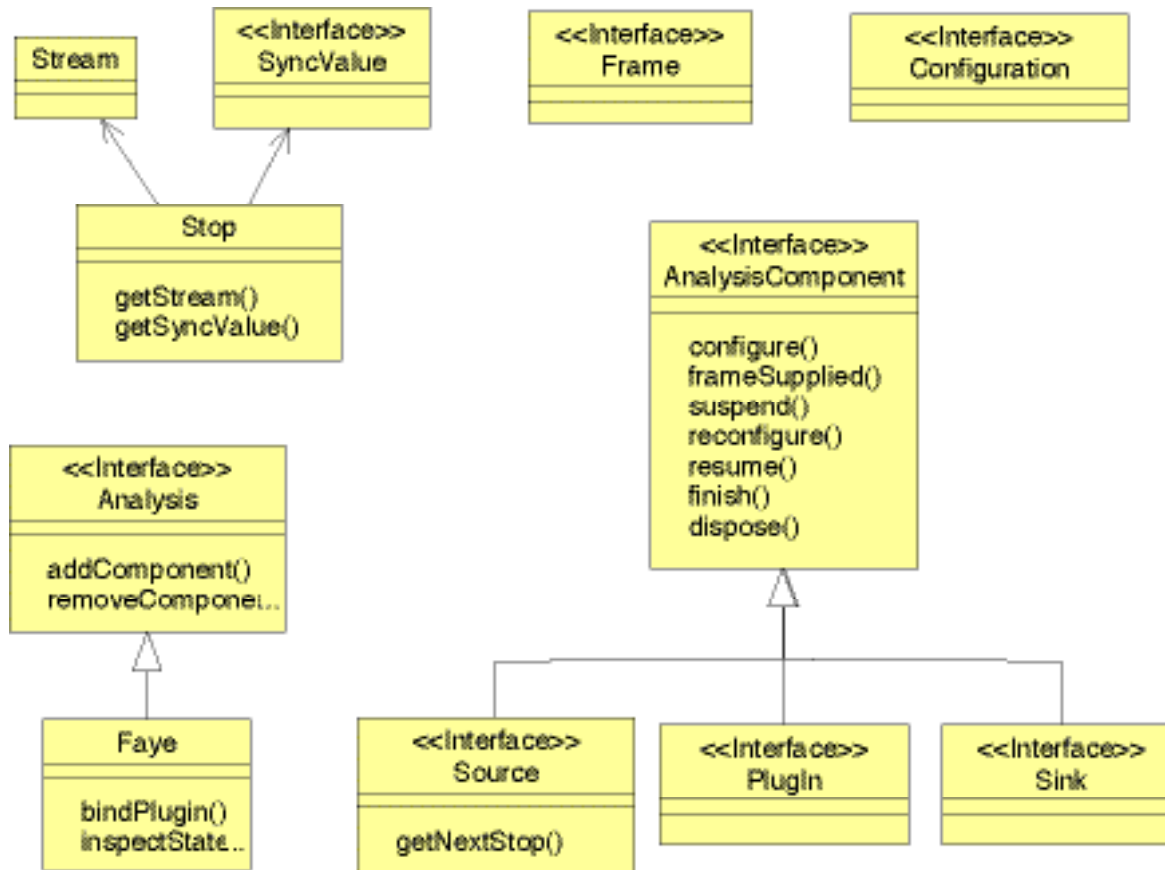
- **Can generalize Source, PlugIn and Sink into “superclass” AnalysisComponent.**



State Diagram for AnalysisComponent



Revised Class Diagram for Faye



Expand and Define the *Responsibilities* of each Class and Interface

- **The key to good Object Oriented design is defining what are and are not the responsibilities of a object.**
- **Can be trivial:**
 - e.g. `Stream` (a dynamic enumeration)
 - Requires and equality operator;
 - Creation or arbitrary new elements;
 - A set of standard elements.
- **or complex:**
 - e.g. `Faye`.

Responsibilities of Faye

- **Start-up**
- **Configuration of components**
- **Order of execution of components**
- **Analysis looping**
- **Status inspection**
- **Re-configuration of components**
- **Termination**

Document Responsibilities in Unit Tests

- **JUnit is a Java implementation of the XUnit framework.**
- **Will not go into details here.**
- **Summary:**
 - Subclass `TestCase` to create tests.
 - Each method whose name starts with `test` will become a test.
 - Standard executables run tests and report the results.

JUnit tests for Stream

```
public void testNotValidEquality() {
    Assert.assertEquals(Stream.NOT_VALID,
        Stream.NOT_VALID);
}

public void testNullInequality() {
    Assert.assertTrue(!Stream.NOT_VALID.equals(null));
}

public void testExtensionEquality() {
    event = new Stream("Event");
    Assert.assertEquals(event, event);
    Stream tmp = new Stream("Event");
    Assert.assertEquals(event,
        tmp);
}

public void testExtensionInequality() {
    testExtensionEquality();
    Assert.assertTrue(!Stream.NOT_VALID.equals(event));
    Assert.assertTrue(!event.equals(Stream.NOT_VALID));
    geometry = new Stream("Geometry");
    Assert.assertTrue(!event.equals(geometry));
    Assert.assertTrue(!geometry.equals(event));
}
```

Implement Methods so that Tests can run Successfully

- **Tests define the API necessary for an object to carry out its responsibilities.**
- **Only implement API needed for tests.**
 - Saves implementing unnecessary methods.
- **Can write Test for interfaces, which are use to test subclasses.**
- **Rerun test whenever code is modified.**

Clean up “package”

- Once you have a set of code which “works”, i.e. passes all the tests, it should be reviewed before “release”.
- Some items to look for:
 - Minimize the `public` part of the API.
 - Use `interfaces` where possible.
 - Make classes `immutable` is practical.
 - Make classes `final` is appropriate.
 - a Java concept.

Summary

- **Set Scope of the Project.**
- **Select Use Cases.**
- **Find Classes and Interfaces.**
- **Characterize Collaborations.**
- **Capture Responsibilities in tests.**
- **Write code to pass tests.**
- **Clean up public API.**