EpicsCoreJava

Gabriele Carcassi, OspreyDCS

Kunal Shroff

VTypes

VTypes

- Java interfaces for values
  - Not tied to protocol or implementation (i.e. we can read them from databases, epics protocol, file, …)
  - Not tied to memory representation (i.e. they can come directly from the network buffer, can be lazily calculated, …)
  - Immutable
- It’s the “just give me some value from somewhere that is safe to read”

VTypes

- Currently only scalars and arrays are implements
  - No VTable yet
- Abstract classes instead of interfaces
  - We wanted to guarantee that basic functionality (i.e. equals / hashcode / toString) has the same implementation and unfortunately this cannot be provided by default methods
    - Previously, we gave some standard implementations that one would have to point to (i.e. VTypeToString.toString), but there was no guarantee anybody did it
  - Also wanted to avoid FrankenValues (e.g. a VDouble that is also a Vtable)
  - We can put static factory methods with the appropriate class
    - VString.of(“mystring”, Alarm.none(), Time.now()) instead of ValueFactory.newVString(“mystring”, ValueFactory.alarmNone(), ValueFactory.timeNow())

VTypes

- Alarm, Time, Display, … through composition (not inheritance)
  - That is, instead of VScalar extending Alarm, it has a getAlarm() method
  - We couldn’t really fully use the hierarchy to know whether a value had an alarm
    - To handle nulls, disconnects and values with no alarm, the generic way to get an Alarm from a value was ValueUtil.alarmOf(value, connected)
    - Now it is Alarm.alarmOf(value, connected) and similar utility methods are static methods of the appropriate abstract class
  - This allows to treat those pieces individually
    - Obvious way to compare them (i.e. value1.getAlarm().equals(value2.getAlarm()) instead of VTypeValueEquals.alarmEquals(value1, value2))
    - May have minor performance improvements (i.e. hopefully most values have no alarm, and that’s the same object, equality just check the pointer, …)

VTypes

- Support for unsigned scalars and arrays
  - In epics.util we added basic classes to fully support unsigned numbers (ULong, UInt, UShort and Ubyte that extend java.lang.Number) and unsigned arrays (ListULong, …)
  - These provide numeric conversions as expected, but when you take the values out in terms of java primitives you get the signed version
    - That is listUByte.getInt(0) will be 255 but listUByte.getByte(0) will be -1
- We then added VTypes that wrap around those classes
  - VLong.getValue() returns a java.lang.Long with VULong.getValue() returns an org.epics.util.number.VULong

VTypes

- Reviewed Alarm to better match Epics 4
  - It has a severity, status and message
- Reviewed Display to use ranges instead of point values
  - Each pair of values is a Range
  - It is easier to check whether the Range is undefined (i.e. display.getAlarmRange() == Range.undefined())
  - It is easier to see whether a value is in a range

Epics-util

Epics-util

- Java utility classes
  - Basic functionality that we'd hope someday to be directly supported in the Java standard library
- Extract them in a single library to be used across all other EPICS Java projects

Unsigned types

- org.epics.util.number provides support for unsigned integers
- ULong, UInteger, UShort and UByte primitive wrappers that extend java.lang.Number
- UnsignedConversions provides all the possible widening conversions (narrowing conversions are the same)
- Currently does not provide mathematical operations, though it can be added if needed

Numeric collections

- org.epics.util.array provides support for numeric collections
  - Allows iterating on a set of numbers regardless of their type
  - Allows creating read-only wrappers
  - Allows implementing arrays that are lazily calculated
  - If using the concrete array implementations, the performance is guaranteed to be the same as using arrays directly
- Includes interfaces/implementations for all 10 numeric types (2 floating point, 4 integers, 4 unsigned integers)

Other bits

- org.epics.util.concurrent provides utility for concurrency
  - Allows to create pools of daemon threads (i.e. they don’t force the JVM to keep running) that well named (simplifying debugging)
  - Implements a queue for batch processing (i.e. items are added one at a time that read in batches)
- org.epics.util.stats provides support for statistical objects
  - Currently only numeric and time ranges
- org.epics.util.text provides support for parsing text
  - For example, provides number formats based on the EPICS 3 and 4 type definitions (i.e. precision, format string, …)

Observations on the pvData api

Writting client code to handle type casting is not trivial

```
long[] pvIndices = new long[value.getSubField(PVScalarArray.class, MasarConstants.P_CONFIG_INDEX).getLength()];
converter.toLongArray(value.getSubField(PVScalarArray.class, MasarConstants.P_CONFIG_INDEX), 0,
        value.getSubField(PVScalarArray.class, MasarConstants.P_CONFIG_INDEX).getLength(), pvIndices, 0);
```

Using Arrays is not intuitive nor easy

```
PVStringArray names = (PVStringArray) request.getScalarArrayField(MasarConstants.F_NAME,
        ScalarType.pvString);
names.put(0, 2, new String[] { MasarConstants.F_SYSTEM, MasarConstants.F_CONFIGNAME }, 0);
```

Epics-util and pvData

Working with direct types

```
// Common superclass for all numerical arrays. Returns a no-copy array wrapper with a generic interface
PVNumberArray pvArray = ...
ListNumber values = pvArray.get();

// Provides index access for any primitive type taking care of casting (including unsigned conversions)
float sum = 0;
for (int i = 0; i < values.size(); i++) {
    sum += values.getFloat(i);
}

// Provides iterator access
IteratorNumber iter = values.iterator();
double sum = 0;
while (iter.hasNext()) {
     sum += iter.nextDouble();
}

// Allows writing from any type to any type
pvArray.put(5, CollectionNumbers.toListDouble(1,2,3));
```

Epics-util and pvData

Working with direct types

```
// Specific arrays return concrete unmodifiable array wrappers. Direct use of the wrapper provide no performance
cost over array (after JIT warmup)
PVUByteArray pvArray = ...
ArrayUByte values = pvArray.get();

// Slicing array just provides a view
ArrayUByte slice = values.subList(3, 5);
// You can always make explicit (and mutable) copies
ArrayUByte sliceCopy = new ArrayUByte(slice);

// The wrapped array is always accessible
UnsafeUnwrapper.Array<byte[]> data = UnsafeUnwrapper.readSafeByteArray(values);
for (int i = data.startIndex; i < data.size; i++) {
    byte b = data.array[i];
}

// You can also ask for a write safe version
UnsafeUnwrapper.Array<byte[]> data = UnsafeUnwrapper.writeSafeByteArray(values);
```

GPClient

What is the generic purpose client?

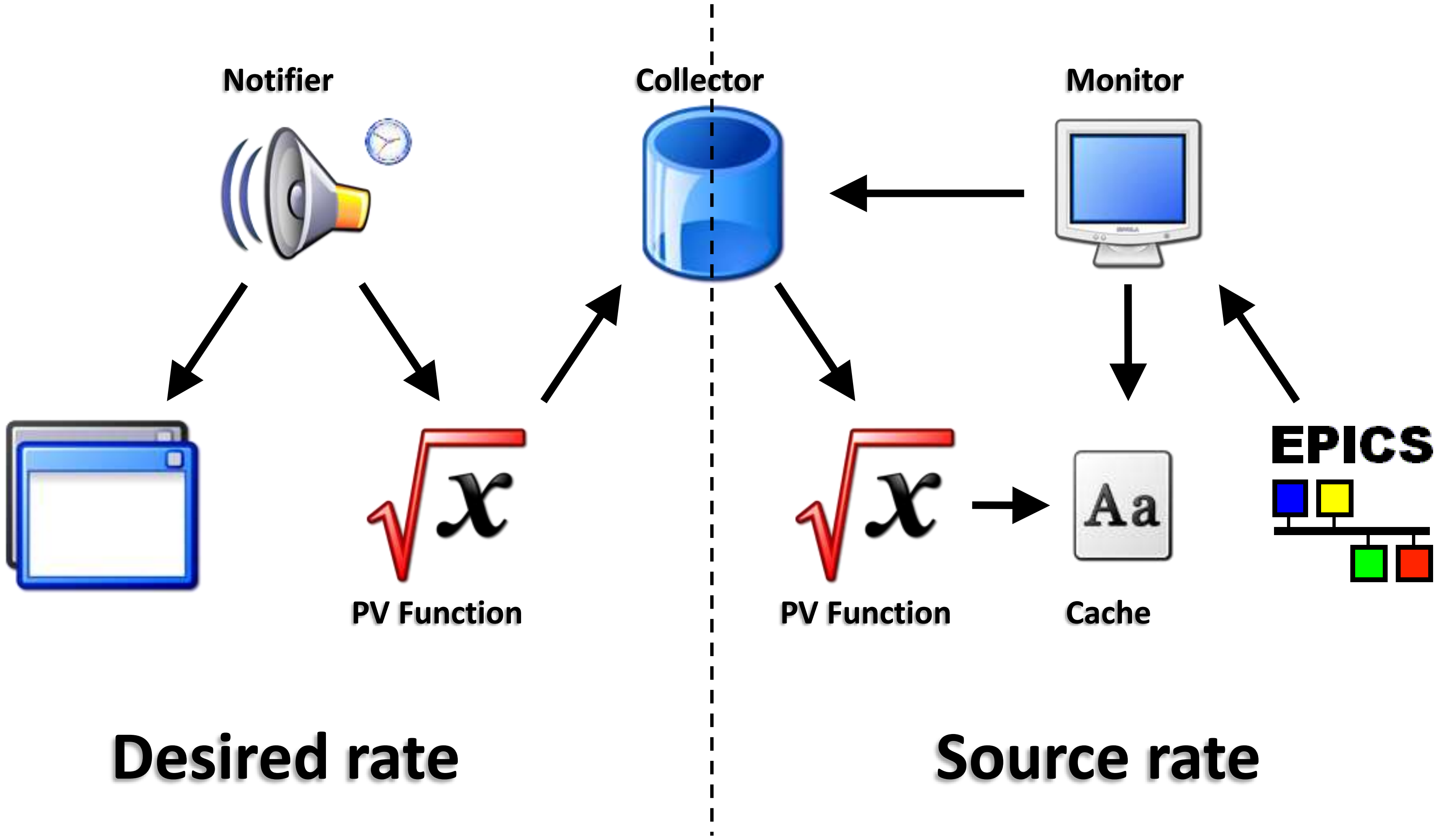
- It’s a better version of pvmanager/dirt
  - Client library to be used when writing a generic purpose application, that does not particularly care about protocol details and does not need an ad-hoc caching and synchronization model
  - It’s a redesign based on the past experience: we kept what was working, removed the parts of the architecture that we never actually used and better generalized the parts that were used
- It’s the “just give me some data and take care of all the threading stuff” client library

```
PVReader<VType> pv = GPClient.read(“pva://MGNT10:CurrRB")
                   .addListener((PVEvent event, PVReader<VType> p) -> {
                        System.out.println(event + " " + p.isConnected() + " " + p.getValue());
                   })
                   .start();
```

Functionality inherited from pvmanager/diirt design

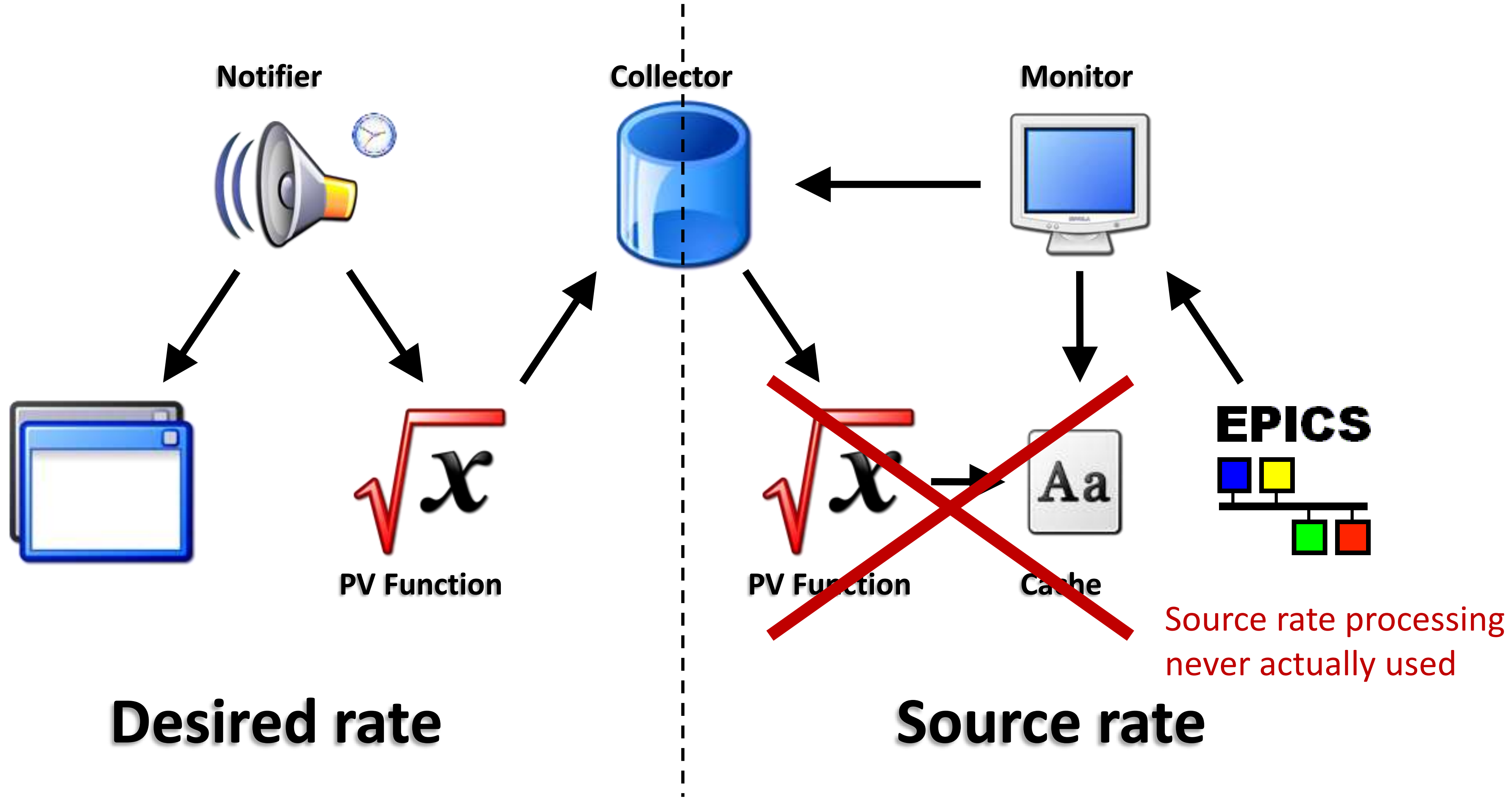
- Extensible architecture
  - DataSources can be added by dropping in new libraries
- Handles event rate decoupling
  - If notifications come in faster than they are consumed, it automatically skips or batches the updates
  - No two notifications in flight
- Thread safety and isolation
  - No notification comes when one is still processing
  - All values are immutable
  - Bad pv listener cannot lock the EPICS protocol library

Changes from pvmanager/diirt design

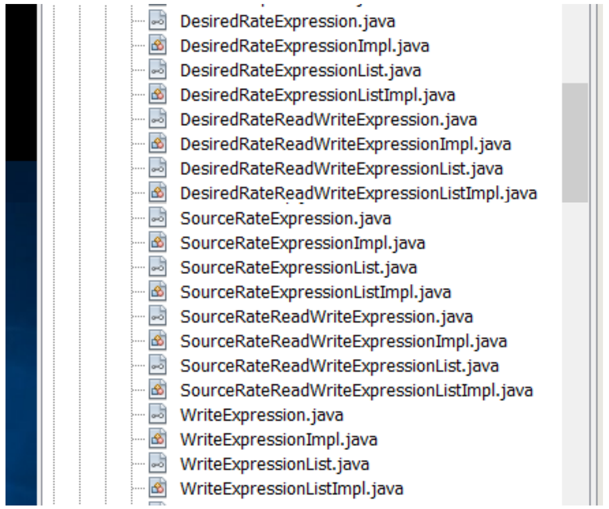


Desired rate **Source rate**

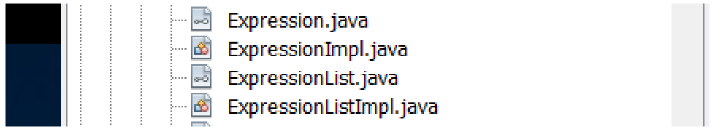
Changes from pvmanager/diirt design



Changes from pvmanager/diirt design



DesiredRateExpression.java
DesiredRateExpressionImpl.java
DesiredRateExpressionList.java
DesiredRateExpressionListImpl.java
DesiredRateReadWriteExpression.java
DesiredRateReadWriteExpressionImpl.java
DesiredRateReadWriteExpressionList.java
DesiredRateReadWriteExpressionListImpl.java
SourceRateExpression.java
SourceRateExpressionImpl.java
SourceRateExpressionList.java
SourceRateExpressionListImpl.java
SourceRateReadWriteExpression.java
SourceRateReadWriteExpressionImpl.java
SourceRateReadWriteExpressionList.java
SourceRateReadWriteExpressionListImpl.java
WriteExpression.java
WriteExpressionImpl.java
WriteExpressionList.java
WriteExpressionListImpl.java



Expression.java
ExpressionImpl.java
ExpressionList.java
ExpressionListImpl.java

The old API differentiated at compile time between:

- Source/Desired rate expressions
- Read/Write/ReadWrite expression
- Single/Group of expression

To handle all cases, there are 20 classes

In the new API:

Source rate expressions are removed

- Never actually used

All expressions are ReadWrite

- The compile time check could not remove the runtime check: may as well remove the compile time one and simplify the API

We currently have only 4 classes

Changes from pvmanager/diirt design

- DataSource recipes
  - In the old API each pv expression created a datasource request with all the channels the pv would connect to, so that the datasource could (in principle) optimize connection to multiple channels
  - In practice: 99% of expression mapped to a single channel and the batch connection was never optimized
  - In the new API each datasource request is for one channel only which simplifies the API and the implementation
    - However, the requests are posted on a processing thread and then batched, so if batch connection is ever implemented it would actually work across pvs
- DataSource write implementation
  - In the old API, the write for a channel had to be implemented synchronously
  - In the new API, the write can either be implemented synchronously or asynchronously, simplifying the implementation of each data source

Changes from pvmanager/diirt design

- Unified callbacks for read and write (ReadCollector and WriteCollector)
  - In the old API, the data source was given a different callback for connection/value/error/writeConnection/writeResponse
  - In the new API, a read request is given a ReadCollector and a write request is given a WriteCollector, and they can be used for all notifications
- Generalized callbacks for read and write
  - In the old implementation, event notification for data sources is ad-hoc. If one wanted a pv to react from events that are non-DataSource generated, this had to be also handled ad-hoc
    - For example, graphene (the graph library) has to respond to data update events and to UI events, and the UI event pipeline had to be implemented separately
  - In the new API the ReadCollector and WriteCollector are not tied to the DataSources and therefore one can digest updates coming from different sources (i.e. UI events, command/response services, …)

Changes from pvmanager/diirt design

- Unified event pipeline
    - In the old implementation, the events for connection/value/error/writeConnection/writeResponse where handled separately in an ad-hoc way, and the client event was yet a different object
    - In the new API, there is a single PVEvent that supports all of the types, and it is the same object for source rate notification and desired rate notification (i.e. it passes through in most cases)
        - That same object supports combining multiple events into a single events. Common events (i.e. connection and value) are actually the always same object (thus saving creating/destroying objects). The event batching preserves ordering (i.e. you can distinguish between "newValue-then-error" and "error-then-newValue" within a single notification)

Changes from pvmanager/diirt design

- Null values are now supported
  - In the old implementation, null values could not be written and could not be received.
    - This was annoying when local pvs where used for selections. The null initialization would mean no selection but, once the value was changed, there was no way to “clear” the selection.
  - In the new API, null value are supported by the framework (though not necessarily by all data sources)
- Channel name only connection
  - In the old API, one would always need to at least specify the maximum rate of events and whether the string represented a channel or a formula
  - In the new API, there is a default maximum rate and a “string only” expression

What still needs to be done

- PVAccess support could be improved with a redesign of the pvaccess library
    - Currently, pvaccess library exposes a cache for the latest value therefore the only safe way to share that data is to make a defensive copy
- Other functionality could be retrofitted from diirt
    - Support for services (command/response)
    - Formulas

Summary

- The EPICS VTypes are a redesign of diirt Vtypes
  - It is part of EPICS core
  - Same idea of interfaces to immutable data
  - All utility methods are static methods of the respective classes
  - All object provide reasonable equals/hashcode/toString
- The EPICS GPClient is a redesign of pvmanager/diirt
  - It is part of EPICS core
  - Continues to provide similar functionality to pvmanager/diirt
  - Simplified core (44 classes instead of 106)
  - Simplified interface
  - More flexible architecture