**Usage:**

java -jar <target/scb-reference-data-publisher-1.0-SNAPSHOT.jar> <data/lme\_upstream\_data.txt> <data/prime\_upstream\_data.txt>

* Main class: com.tianjue.referencedatapublisher.ReferenceDataPublisher
  + It reads the files and the replay the data to simulate the upstream publishing
  + it supports sleep

**Thread Model:**

* **upstream channel**
  + **upstream channel**
    - LME exchange/PRIME broker data source
  + **Listener**
    - message listener – it parses the raw message and pass to core processor
    - it is registered in the channel
  + **the application doesn’t really define the thread model in the upstream**
    - it could be one thread per channel
    - it could be one thread for multiple channels (uses Selector)
    - TestFileChannel is to use to read the files and itself contains a thread. So each file is processed by one thread – for simplicity
  + **Separation of concern:**
    - decouple the communication mechanism from message process
    - decouple different type of data source communication and process, as they can be varied a lot among sources (LME/Prime)
* **core processor**
  + core logic to apply the business rule
  + the threads are created by executor package
    - the thread number can be configured in SystemConfig
  + all the updates for same MATCHING CODE will always be processed in the same thread to reduce data race risk
    - the routing mapping is String -> int
    - the implementation is using String hash code for routing logic for simplicity
      * in practice, the routing logic can be grouped into different code set
        + few high liquidity instruments can be processed in particular thread
        + some thread can process more less liquidity instruments
* **downstream api**
  + Component API
    - the application registers the subscriber to SubscriberManager
  + publisher uses parallel stream as each subscriber is independent
  + we can define customized fork join pool to execute if further control is required

**Instrument Data:**

* **InstrumentData**
  + instrument data when receiving from sources and pass through processors
  + the subclass according to different source can behave differently
  + The assumption format is **Field1=Value1|Field2=Value2|**
  + the data structure is stored as key value Map internally
* **AggregatedInstrument**
  + aggregated instrument for the consumer view

**Business Rule:**

* **Match Code rule**
  + rule to get the primary key for aggregation instrument
  + it can be in a group of rules
  + the first rule will be executed first, if it gets value, it will return. Otherwise, it looks for the new rule
* **Create Aggregated Instrument Rule**
  + rule to create the Aggregated View from publish data
  + only one aggregated rule
* **Aggregation Rule**
  + rule to aggregate the arriving data into the aggregated view
  + It could be in a group of rules
  + The rules in the groups are executed in sequentially
* **Rule Execution Logic**
  + utility class to control the behaviour of multiple rules
* **All the above rules can be combined with flexibility**
* **For future improvement, those rules interface can have onException parameter to handle exception, like listener.**

**Config:**

* define different set of registered functions
* new upstream can define new sets of registered functions
* assume they are set in initializations and they are not thread safe for simplicity

**Behavior of Console application:**

