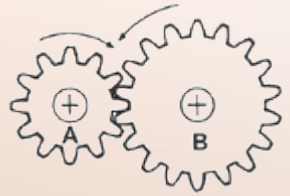
JRTE

Rational Transduction Engine for Java

A pattern-analytic approach to text data mining

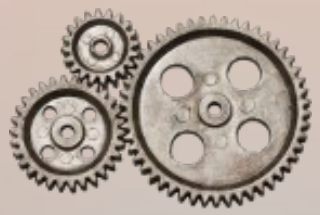
Outline



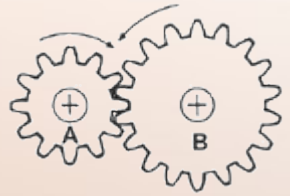
- Overview of JRTE
  - History, current status, direction
- Pattern analysis and text extraction at Q1 Labs
  - Use cases for JRTE in security contexts
  - Transducer design and development
- Q&A, JRTE demonstration
  - JRTE performance benchmarks
  - Linux kernel log processing
- Wrap up

JRTE

Overview

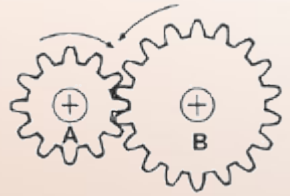


What is JRTE



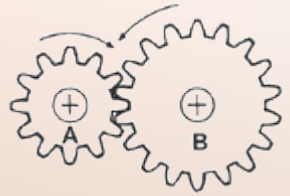
- JRTE is a framework for applying finite state transducers to various problem domains
  - Information extraction from text data sources
  - Rule-based behavioral monitoring and governance
  - Two-way information exchanges
- High performance runtime engine
  - Supports multiple concurrent transductions
  - ~70 MB/s text pattern recognition throughput
  - ~50 MB/s text pattern recognition + extraction

Components



- JRTE runtime transduction engine
  - Transduction factory binds gearbox to target
  - Application provides input and runs transduction
- JRTE gearbox compiler
  - Compiles automata produced by ginr to transducers
  - Packages transducers into gearbox for runtime use
- ginr regular expression compiler
  - Algebraic manipulations of regular expressions
  - Compiles these to automata for gearbox compilation

Current Project Status

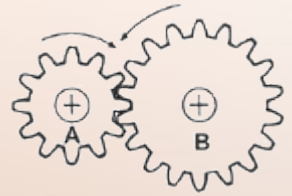


- JRTE is a functional open source (LGPL) product
  - Extensible, easily adapted to specific applications
  - Hosted at http://code.google.com/p/jrte/
- Base framework includes support for text extraction
  - Selective extraction and embellishment of input
  - Assignment of extracted text to named values
  - Extensions assemble domain objects from values
- Needs real-world problems to evolve

JRTE

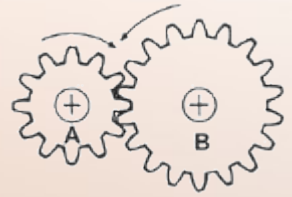
JRTE@Q1Labs

Use Case: Log Analysis



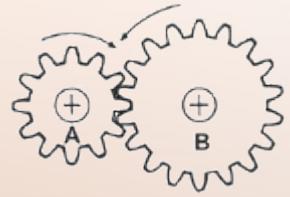
- *Client has a large aggregate volume of inbound logs containing information that is required to be selectively extracted into domain objects*
  - Many different log sources, diverse formats
  - Volume may exceed 10 MB/sec
  - Logs may be processed ”off the wire” in real time or streamed to disk for near real time processing
- Solution: Use ginr + JRTE to define a transducer for each distinct log format

Use Case: Rapid Development



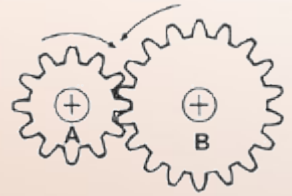
- *Client has a substantial team of developers devoted to log management and wishes to reduce cost of development and development time*
  - Has tried several approaches to ease development but no silver bullet
  - Java parsers for complex log formats are difficult to write and maintain
- Solution: Use ginr + JRTE to build log parsers with minimal Java coding

Use Case: Reduce System Costs



- *Client wants to reduce hardware costs and pass savings on to customers*
  - Reduce CPU and memory footprint for log management
  - Increase throughput on machines dedicated to log management
  - Reduce the number of potential failure points in the system
- Solution: Incrementally migrate log management infrastructure to JRTE to enable a larger volume of log input to be processed per machine

Use Case: High-level Language

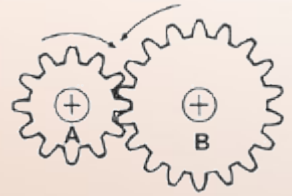


- *Client wants to enable customers to define their own log parsers*
  - Enable customers to define log parsers and integrate them into the system without waiting for client development team
  - Client has existing high-level grammar to enable this
- Solution: Use ginr as intermediate compilation target for the existing high-level grammar, and use JRTE to compile ginr outputs for integration into the runtime system

JRTE

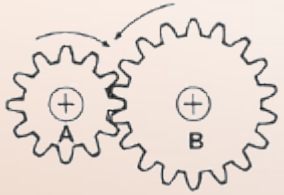
Design & Development

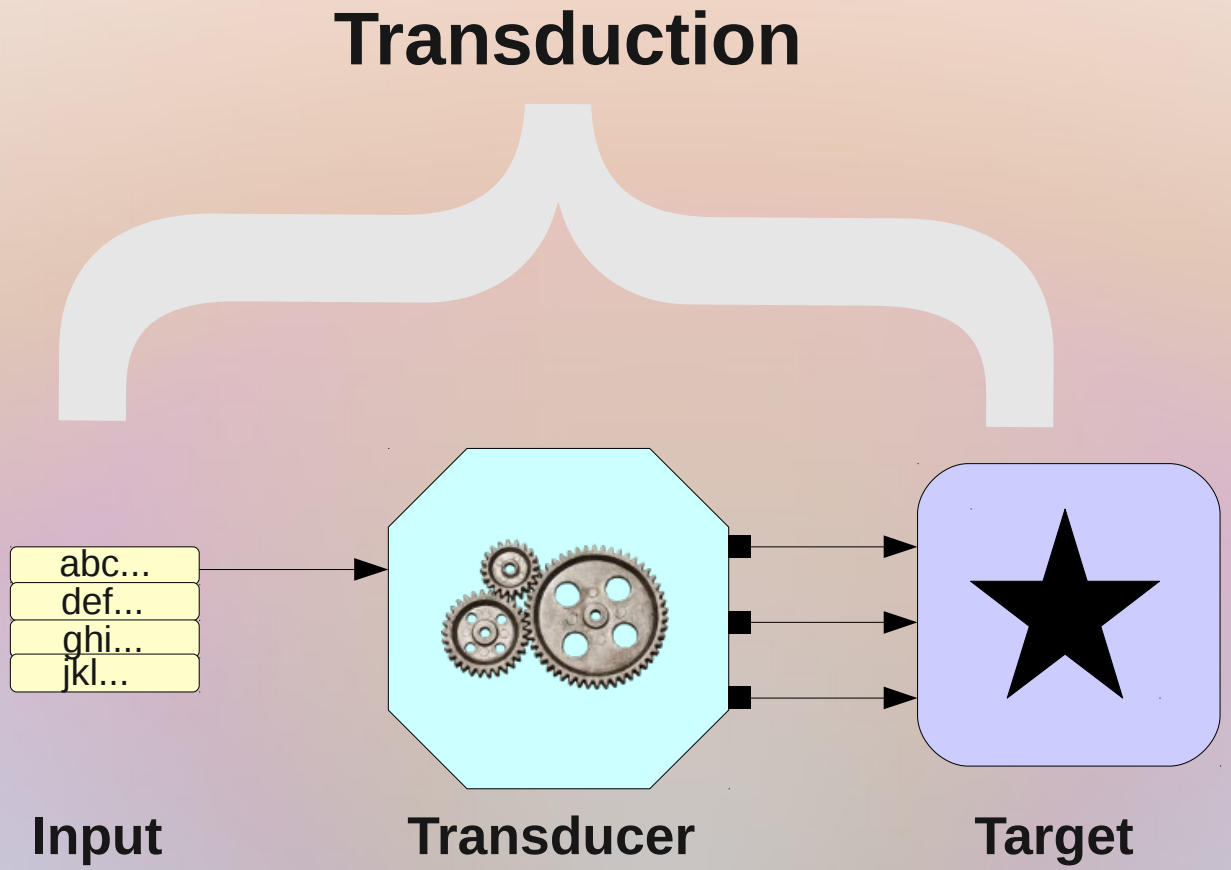
Some Definitions...



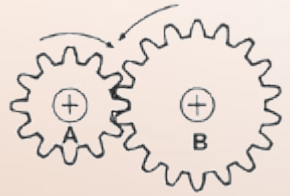
- *Transduction*: main JRTE component consists of an input source, a transducer, and a target object
- *Input source*: a stack/queue of sequential streams of characters or signals
- *Transducer*: a set of rules compiled from a regular expression that relate inputs to target effectors
- *Target*: An application-defined Java class that encapsulates application business objects
- *Effector*: a lightweight Java class that acts on the target

Transduction: Graphic



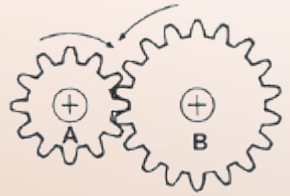


Input Patterns



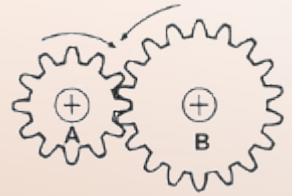
- Patterns are ternary regular expressions that define how inputs relate to target effectors
  - `((a, paste)* (b, cut[buffer]))* (eos, stop);`
  - Recognizes input patterns matching `(a* b)* eos`
  - Invokes target effectors `(paste* cut)* stop`
  - Effectors are invoked in lockstep with inputs
- Main design task is to extricate the input pattern (without reference to target effectors)
- Input pattern can then easily be extended to include effectors, completing transducer definition

Input Streams



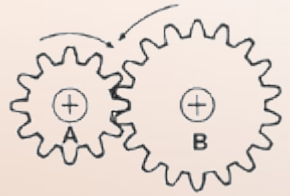
- Input classes implement IInput interface
  - `public CharBuffer get() throws InputException;`
  - `public void mark();`
  - `public void reset() throws InputException, MarkLimitExceededException;`
- Implementation classes wrap standard Java I/O classes
  - `public SignalInput(char[][] inputs) throws InputException`
  - `public ReaderInput(Reader input) throws InputException`
  - `public StreamInput(InputStream input, Charset charset) throws InputException`

Transduction Targets



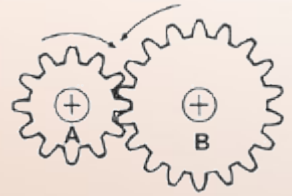
- Targets implement ITarget interface:
  - `public String getName();`
  - `public IEffector<?>[] bind(ITransduction transduction);`
  - `public ITransduction getTransduction();`
- Base Transduction class implements ITarget and provides basic effectors for text extraction and transduction control
  - `paste, cut, copy, clear, out, outln, save, restore`
  - `start, stop, shift, in, mark, reset, pop, pause`
  - `count, counter`
- Applications extend this by implementing ITarget with custom effectors to construct domain objects from extracted values

Target Effectors



- Lightweight anonymous inner classes
- Simple effectors implement IEffector interface
  - `public int invoke();`
- Parameterized effectors implement IParameterizedEffector
  - `public int invoke();`
  - `public int invoke(int parameterIndex);`
  - `public void newParameters(int parameterCount);`
  - `public int setParameter(int parameterIndex, byte[][] parameterList);`
- All effectors receive a reference to the target object

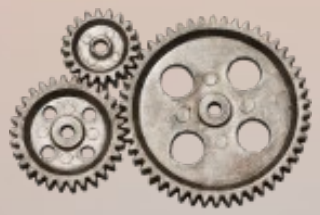
ITransduction



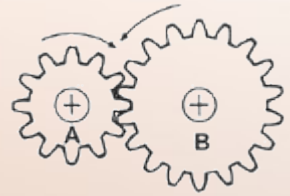
- ITransduction interface represents main runtime integration point
  - `public void start(String transducer) throws RteException;`
  - `public void input(IInput[] inputs) throws RteException;`
  - `public void run() throws RteException;`
- Provides accessor methods for extracted named values
  - `public String[] listValueNames();`
  - `public int getValueNameIndex(String valueName) throws TargetBindingException;`
  - `public String getNamedValue(int nameIndex) throws TargetBindingException;`

JRTE

Build & Deploy

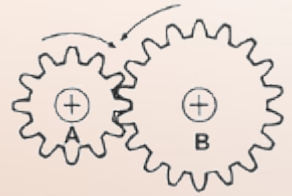


Compiling Transducers



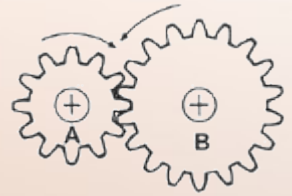
- Transducer definitions are expressed in text files and compiled with ginr to produce intermediate automata
- Standard prologue defines common character classes and intermediate transducers
  - Character classes letter, digit, punct, space, tab, cr, nl, …
  - Use HTML entities to express 8-bit or multibyte characters
  - Character superset 'any'
  - PasteAny = (any, paste)* – typical intermediate transducer
  - 'abc' @@ PasteAny == (a, paste) (b, paste) (c, paste)
- Use ant exec task to run ginr against transducer definitions

Compiling Automata



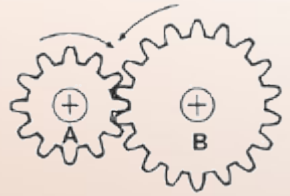
- Transducer automata produced by ginr are compiled with JRTE to produce a runtime gearbox
- JRTE produces more compact automata by finding equivalent input characters/signals classes
- Final automata are packaged into a gearbox for runtime use
- Use ant java task to run JRTE compiler against ginr output automata

Deploying the Gearbox



- The gearbox is a binary file produced by the JRTE gearbox compiler
- Gearbox is a namespace encapsulating transducers, Unicode characters, signals, effectors, effector parameters, and named values
- Gearbox is bound to the target class definition – multiple instances of the target class can be bound at runtime as required
- Jrte runtime class loads gearbox and acts as ITransduction, IInput factory

Jrte Runtime Factory



- Jrte runtime class loads gearbox and acts as ITransduction, IInput factory
  - `public Jrte(File gearboxPath, String targetClassName) throws GearboxException, TargetBindingException`
  - `public ITransduction transduction(ITarget target) throws RteException`
  - `public IInput input(char[][] signals) throws GearboxException, InputException`
  - `public IInput input(Reader infile) throws GearboxException, InputException`
  - `public IInput input(InputStream infile, Charset charset) throws GearboxException, InputException`

JRTE

Q&A + Demo