Using Strings to Compose Applications from Reusable Components

BeComm Corporation
info@becomm.com
October 4, 2001

The past decade in technology has evidenced an explosive growth in the proliferation of new web and network services, digital data and content, new and increasingly diverse computing devices and new wired and wireless networks for data communication. With the billions investments in hardware infrastructure, software infrastructure and network infrastructure, not only do end users want to see new applications, but ISVs and OEMs must deliver these applications in a way that assuages managements demands for an ever-increasing return on investment. As a result, this changing landscape has imposed a critical challenge on software developers and IT managers to improve time to market and reduce development costs. To leverage this existing investment in hardware, network and software infrastructure, applications must be developed to capitalize on new revenue opportunities while also reducing existing operating expenses. The key observation made by many is that reuse of technology assets to compose new applications is critical to addressing these challenges.

There are, however, several issues that make reuse difficult to achieve. First, reuse is an easily misunderstood concept, as it is not simply using previously engineered or acquired technology asset more than once. It requires reuse engineering that prepares technology assets to be reusable. Second, identifying what software components can be reused is a confusing process, as traditional design approaches tend to prevent reuse outside the narrow scope or domain in which they were initially developed. Finally, software engineering techniques and methodologies alone are insufficient tools for developers to achieve true reuse. Even with the best development methodologies and architectural techniques, taking a traditional approach to development can reduce the potential for reusing technology assets or limit the type of potential reuse, without later additional re-engineering.

To address these issues requires an application framework that promotes reuse at every level: from fine-grained application components to large-scale back-end systems. The lack of a solid application framework for reuse has prevented it from being widely accepted and implemented. Such a framework needs to yield solutions that are *dynamic* enough to adapt reusable components to changing networks, media types, and device types at runtime in unanticipated ways; *distributed* so as to leverage services within the WAN and LAN; and *efficient* enough to run on network servers and resource-constrained embedded devices alike. While traditional techniques (e.g., object-oriented design and component systems with the appropriate extensions to support client/server systems) and methodologies (e.g., UML that aids in the specification and design of systems) have various strengths they have two inherent limitations that prevent the developer from creating solutions that are truly reusable. These limitations are as follows:

Limitation 1: Imperatively Defined Configuration Intelligence

In traditional application development, the software developer explicitly identifies a set of required services and specifies how and when to interface with them. This necessitates that developers have a priori knowledge of all services that are to be used directly by their component. This entanglement of configuration intelligence with application logic transcends from the so-called “main loop” of the application down to granular components composing the application. While conventional OO/component techniques enable the developer to be oblivious of an algorithm’s implementation details, they are still required to know the interfaces, and to write code that depends on those interfaces to meet the goals of the application. Consequently, it is difficult to chain together services to achieve a desired goal without explicitly specifying all possible configurations in advance. Therefore, in practice, imperatively defining the configuration intelligence limits the application to a static set of predefined interactions.

Limitation 2: Process-centric Computing

The traditional notion of an "application" implies a process-centric viewpoint, in which each application is embodied by a computational process on a node in the network. From the end-user's perspective, the application is usually perceived to be the process with which they are directly interacting (e.g. a spreadsheet or a web browser), possibly extended to include the back-end services supporting that process, such as a web server and CGI program. In the case of distributed systems, the "application" might be interpreted as living on both the client and server node.

The framework for enabling communication is considered external to the application itself, existing as a separate layer on which the application resides. Application development focuses on defining the computational process, encoding the relationships between objects, and invoking methods from a known set of interfaces, but the flow of data between processes is considered subordinate to the processes themselves. This "process-centric" notion of an application is inherently limiting to the developer, because it does not easily accommodate the potential for an application to participate as a service operating on a stream of data flowing through a dynamic configuration of other services.

Today's emphasis on return on investment requires the reuse of legacy technology assets as well as newly designed algorithms, and therefore evidences the need to overcome traditional application design limitations such that application logic can be reused in a more flexible, efficient, and cost sensitive manner.

Strings™ defines an application framework that supports build-time and more importantly runtime adaptability of reusable software components. The methodology used to develop Strings-based applications facilitates the creation of software components as services with an unprecedented degree of reusability, thereby enabling software developers to quickly create smarter and less expensive solutions.

2 Strings Methodology

To achieve an unprecedented degree of reuse, the Strings methodology is to (i) turn application logic into services (ii) to augment the process-centric paradigm with a dataflow-centric paradigm, in which the defining principle of the application are goal rules that control the flow of data itself over these services. The data can then pass through an ad hoc defined sequence of reusable application services. The fact that these services may be embodied in processes is secondary to the primary notion of a stream of data.

By separating configuration intelligence from application logic into rules and primitive services, respectively, it is possible to declaratively define how and when the service can be used. This separation results in a form of communication indirection, in which no service refers to the interface of any other service, and no two objects talk directly to each other. Instead, an "intelligent engine" facilitates the creation of communication paths between objects automatically by reasoning about the goals of the application.

In this way it is possible to hook services together at run-time as they are discovered. As a result, the traditional tight coupling of objects and the so-called "main loop" of the application ceases to exist, being replaced by a set of rules and a "higher-order control" that coordinates the flow of communication between services. With such an approach, applications no longer exist as static "programs", but rather as dynamic invocations of services that can be automatically configured in unanticipated ways, adapting to meet the goals of the system on the fly.

The resulting inversion of control greatly improves the task of building applications, as the programmer is freed from the task of specifying configuration information imperatively. Routing decisions and state management are removed from the application, and handled by the system. In this model, objects can talk to each other without a priori knowledge of each other's existence.

3 Strings Framework

The Strings framework consists of the following components:

1. A dynamic set of discrete services, called *Beads*™, which encapsulate application logic.
2. A set of declarative rules specifying when beads are used to achieve some application-specific goal, which can either be specified by the user, system's integrator, or catalyzed by system events at runtime.
3. An "engine" that matches the application-specific goals with the declarative rules of the beads, and thereby decides what services to employ to meet the goals.

The rest of this section discusses this components in further detail.

3.1 Beads

Beads encapsulate resources as discrete services that adhere to a highly structured interface. The bead itself is a software component that exports a granular unit of functionality. Examples of resources that a bead can encapsulate include:

- hardware such as a video display, speaker, microphone, mouse, Ethernet, etc.
- protocols such as TCP/IP, HTTP, SOAP, email (POP3, SMTP), etc.
- transformational algorithms such as audio/video decoders, etc.
- SDK technologies such as speech-recognition engines (e.g., IBM's ViaVoice), text-to-speech generators, etc.
- Backend services such as Database, CRM, and Content Management Systems.

In other words, any resource that can be defined as a service can be encapsulated within a bead. There is no direct reliance on a particular programming language (such a Java) required to use the framework. The purpose of the bead is to completely hide both the interface and the implementation of the resources they encapsulate, and thereby cleanly separate the internal algorithm from its external relationship to other such algorithms. The methodology of encapsulating functionality into discrete elements encourages a software partitioning that exhibits no redundancy or dependencies on other beads. This results in code that is highly reusable, and enables services to be assembled into dynamic applications by an automated engine.

Beads are the building block for the pipeline of processing elements applied to a data flow. Beads conform to a strongly-typed interface pattern, in which operations are represented as *Edges*, which are the touch points of how beads are strung together into a pipeline. Configurations of pipelines range from linear, unidirectional to complex, bi-directional flows of data. Bi-directional flows are automatically managed by the Strings engine, such that a bead may send data in the opposite direction of the flow without needing to know from where the flow originated.

A bead is described to the Strings engine in an XML-based *Bead Schema* that declares the number and direction of its edges and the conditions indicating when its input edges can be used and what type of data comes out of its output edges. A Bead Schema can be authored directly in XML or with the *Bead Schema Editor*, which is a development tool included with the Strings toolkit.

Figure 1 shows an example bead schema that defines the operation of an MPEG audio decoder. The schema identifies the bead's name, description, and one or more edges. Each edge is a separately nameable entity, and in this example the MPEG decoder bead has a single "filter" edge, which logically is both an input and output edge. The precondition of the edge predicates that it should only be used if the content-type variable of the data is set to the MIME-like string of 'audio/mp3'. Similarly, the post condition overrides the content-type variable to the MIME-like string of 'audio/pcm' to indicate the edge's output type.

```
<BEAD_SCHEMA name="mp3decoder">
  <DESCRIPTION>
   The Mp3Decoder bead decodes mp3 data to PCM audio.
  </DESCRIPTION>
  <EDGE name="Decode" shape="filter">
    <PRECONDITION  value="query:Content-Type==’audio/mp3’"/>
    <POSTCONDITION value="namespace:Content-Type=’audio/pcm’"/>
  </EDGE>
</BEAD_SCHEMA>
```

Figure 1. Bead Schema for an MPEG Layer-3 decoder.

Each edge is implemented as a set of handlers: the *message handler*, which is called on to process each message passing through the edge, and several auxiliary handlers to support the creation and destruction of data flows. The partitioning of logic into these handlers, combined with the strongly-typed pipe-like structure of the edge interface, is precisely what enables Strings to hook beads together on the fly to create higher-level services based on the needs of the network, the system, the media types being handled, and user preferences.

3.2 Declarative Rules

Rules are the primary mechanism for configuring Strings, as they define when beads are used to perform some task to achieve some application-specific goal. A rule is defined as a sequence of one or more *steps* to execute when a specific set of *constraints* are met.

Constraints are defined using an evaluation grammar that supports an arbitrary set of evaluation object types. The type of an evaluation object can be an integer, an IP address, a date, or the content-type of the data, etc., which allows for a rich set of declarative rules to be used. For example, if developing an application for the consumer space, it would be possible to declare the constraints of a rule that specify to play music on the home theatre system after 6:00PM. The result of a constraint evaluation determines both whether Strings will execute the steps specified by the rule, as well as which steps to take if multiple rules match the constraints.

If all of a rule’s constraints are met, then Strings will execute the specified steps. The collection of steps associated with a rule is referred to as a *route* along which data will flow. There are three types of steps that the Strings engine can take: 1) a *bead* step which passes data to a particular bead; 2) a *seed* step which populates the “namespace of an application context”[*] with a value that may satisfy some condition of another rule; and, 3) a *loopback* step which specifies what should be sent in the opposite direction of the original data flow.

A rule is described to the Strings engine in an XML-based format that declares a predicate and an associated route of steps. The rules can be authored directly in XML or with a graphical Rule Editor.

Figure 2 shows an example rule that plays audio content of type MPEG to a speaker device, as defined by its predicate and only a single seed step. The rule declares that for data labeled with a content-type of ‘audio/mp3’ the system should seed the application context’s namespace[*] with the value “Target-Device=’speaker’”. It would then be up to the system to determine how to satisfy the application-specific goal of rendering MPEG audio to a speaker. This style of rule is referred to as a goal rule, because it does not specify an “edge step” to pass data to a particular bead.

[*] The application context’s namespace will be further discussed in section 3.3.

```
<RULE>
 <PREDICATE value="query:Content-Type==’audio/mp3’" />
 <ROUTE>
  <STEP>
    <SEED value="namespace:Target-Device=’speaker’" />
  </STEP>
 </ROUTE>
</RULE>
```

Figure 2. Example rule that plays MP3 audio to a speaker device.

Achieving the rule's goal specified in Figure 2 assumes that the Strings engine is configured with additional rules and bead schemas. Let's assume this configuration information comes from the previously described MPEG decoder bead schema shown in Figure 1 (section 3.1) as well as the rule and bead schemas shown in Figure 3, which defines how to render audio data to a speaker device. The rule in Figure 3 declares that the speaker bead's "Encode" edge should be used when the target device is set to speaker, while the bead schema defines that the speaker bead can only accept input of content-type 'audio/pcm'.

```
<RULE>
 <PREDICATE value="query:Target-Device==’speaker’" />
 <ROUTE>
  <STEP>
   <BEAD name="Speaker" />
   <EDGE name="Encode" />
  </STEP>
 </ROUTE>
</RULE>

<BEAD_SCHEMA name="Speaker">
  <DESCRIPTION>
   The Speaker bead delivers PCM audio to an audio output device.
  </DESCRIPTION>
  <EDGE name="Encode" shape="input">
    <PRECONDITION value="query:Content-Type==’audio/pcm’"/>
  </EDGE>
</BEAD_SCHEMA>
```

Figure 3. Example rule and bead schemas to render audio data to a speaker device.

When a message is injected into the system containing MPEG audio data and the content-type is set to audio/mp3 in the application context's namespace, then the Strings engine would evaluate the above mentioned rules, create a pipeline consisting of the MPEG decoder and the speaker bead, and route the message along the appropriate edges of these beads. The outcome is the desired effect of rendering MPEG audio to the speaker device.

The flexibility of using a declarative style configuration system was only partially shown in the above example. A similar "program" could have been constructed straightforwardly using a traditional application design approach that imperatively defines how to decode MPEG audio into a format that can be rendered to a speaker. However, such an application and its components can only be used in the parochial scope for which it was developed. It would be difficult to reconfigure this "program" in unanticipated ways without a fair amount of re-engineering. For example, consider the amount of re-engineering required if the audio data should be processed by a speech-recognition component, and that its output is then sent to either a word processor or an email client. With Strings this is a straightforward process of modifying or replacing the declarative rules, which can be done either at build-time or more importantly at runtime in response to a changing user preference or catalyzed by system events. The flexibility of Strings will be further explored later in section 4.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.