

Collage: A Declarative Programming Model for Compositional Development and Evolution of Cross-Organizational Applications

Bruce Lucas, IBM T J Watson Research Center (bdlucas@us.ibm.com)
Charles F. Wiecha, IBM T J Watson Research Center (wiecha@us.ibm.com)

Introduction: motivation and goals

While the trend towards loosely-coupled inter-networked software is inexorable, our programming models and runtime systems were largely designed for building monolithic, freestanding applications. While the web has vastly increased the scale of distribution, web applications are currently programmed and deployed in a manner not that different from mainframe applications of the 1960s.

We believe that the mismatch between programming models/runtimes, and the predominance of inter-networked software is becoming a significant impediment to producing and composing reliable application software in a timely manner — and that this mismatch is at the heart of much of the dissatisfaction developers have expressed with the complexity and obscurity of current middleware, programming models, and development tools.

The goal of the Collage project is to design a radically simplified declarative programming model and runtime expressly targeted at building and deploying cross-organizational software as compositions of web components. An additional goal is to support an evolutionary style of software development that allows rapid application prototyping, but also enables progressive refinement of the initial prototype into a hardened asset.

In the sections below we outline in more detail a simplified and uniform set of Collage language features addressing end-to-end application design, including business objects, user interaction, and "last-mile" issues such as device adaptation, and multi-modal and multi-device interaction. The discussion below will focus on these aspects of Collage:

- **Simplification.** We provide a single declarative language to support enterprise, web and client applications, hence reducing complexity from multiple redundant abstractions.
- **Evolution.** The RDF-based approach to the programming model used in Collage does not impose encapsulation as strictly as in traditional OO languages. We are exploring whether this more "white box" approach to components increases their flexibility for subsequent reuse without requiring refactoring or redesign.
- **Composition and Distribution.** Our hypothesis is that cross-organizational composition, and the attendant distribution of components, is easier using declarative data-driven programming models than using procedural or object-oriented languages.
- **Device adaptation.** Dealing with the variety of end-user interaction devices has long been a challenge for UI frameworks. Our approach is to reuse a uniform set of programming concepts throughout an application, from back-end to front-end, rather than to use an ad-hoc framework specific to this "last-mile" problem.

Data model

The Collage data model is built on a core subset of RDF. Collage uses the key RDF concepts of *resource*, *triple*, *property*, and *class*. RDF concepts such as sub-property and sub-class play a role in Collage as well. Collage uses RDF resources identified by URLs to build a fundamentally distributed data model.

RDF supports multiple and dynamic classification: a resource may have more than one class, and the classification(s) of a resource may programmatically changed at runtime. These features play an important role in Collage, for such diverse purposes as flexible cross-organizational composition of programs and processes, user interface styling, device adaptation, and so on.

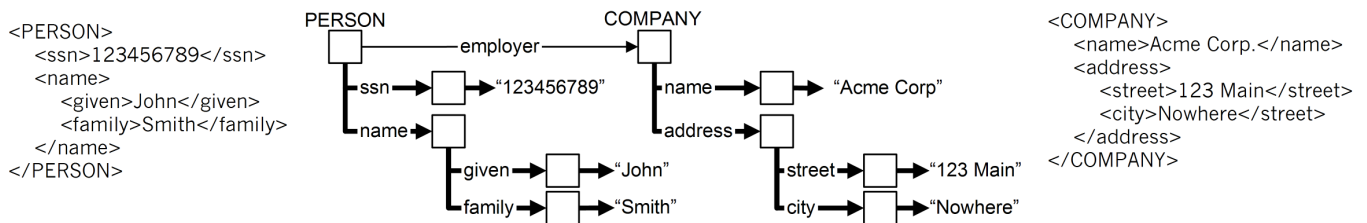
The Collage execution model centers (as do many execution models) around a notion of *mutable entities* with composite values that may be *read*, and whose values evolve over time by being *updated*. The RDF data model underlying Collage lacks such a notion, so the Collage data model builds on the RDF model by introducing the concept of resources with associated composite *values* to model mutable entities. The value of a resource R is

represented by a tree of RDF resource nodes and triples rooted at R. Thus RDF triples are used in Collage both to model a tree-structured composite value of a resource, and to connect related resources in an arbitrary graph. The triples comprising the value of a resource are distinguished from triples that connect resources by having a predicate that is a subproperty of a distinguished property, `c:value`, defined by Collage.

A unifying data model. Collage builds on RDF to provide a data model that unifies several data models and related execution models in common use. The following table describes the mapping between Collage/RDF concepts and each of several popular data models.

Collage/RDF	Entity-relationship	UML	Relational	XML
class	entity class	class	table	–
resource	entity instance	object	row	element, attribute
value property	attribute	attribute	column	parent-child relationship
value tree	composite attribute	–	–	XML (sub-)tree
non-value property	relationship	association	PK/FK	–

XML data. The Collage value tree associated with a particular resource may be viewed as an XML document, using a mapping consistent with the RDF/XML serialization of RDF. In Collage the URI of a resource is a URL, so the the XML document that represents the value of a Collage resource may be read or updated using HTTP GET and PUT operations with text/xml message bodies. The following figure shows two XML documents and their Collage/RDF representations. Boxes represent Collage resource nodes. Lines connecting the boxes represent triples. Heavy lines are triples that form the Collage value tree of a resource. The class(es) of a resource node are noted above the node.



Relational data. Each row of a database corresponds to a Collage resource whose value is the columns of that row. Updating the row corresponds to updating the value tree of the corresponding resource. This is illustrated in the End-to-end Example below.

Execution model

Collage is a programming model for building *reactive systems*. Thus the Collage execution model defines how the runtime state of a program changes in reaction to the occurrence of external events, such as user inputs, messages from external systems, or asynchronous internal events.

The Collage execution model is *data centric*, meaning that all aspects of the execution model are described in terms of changes of the runtime state of the program. The runtime state of a program comprises a set of resources and triples contained in a *triple store*.

Collage provides constructs that the programmer uses to declaratively specify updates to the runtime state. The `<bind>` construct declaratively specifies how the value of a resource changes in response to changes in the values of other resources, using XQuery as a functional value computation language. The `<let>` and `<create>` constructs declaratively specify the creation of new resources, relate new and existing resources by creating and destroying triples, and change the classification of new and existing resources, all based on current resource values.

Execution in Collage is driven by *updates*. An update is the assignment of a value to a resource. Each external event, such as a user input, is reflected in collage as an *initiating update*. In response to an initiating update, a Collage program performs an *execution cycle*. An execution cycle performs a *cascade of ensuing updates*, as declaratively directed by the `<bind>` constructs of the program. After performing the resource value updates directed by the `<bind>` constructs, the Collage program creates new resources, resource relationships, and classifications, as declaratively directed by the `<let>` and `<create>` constructs.

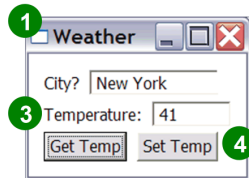
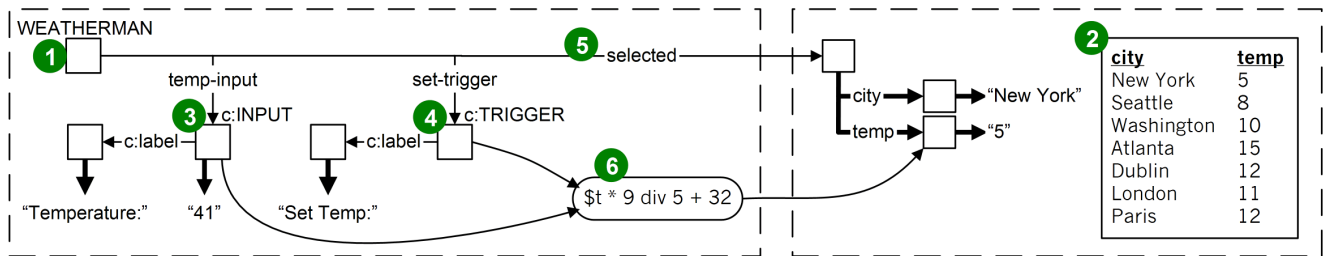
During any given execution cycle, each resource that will be updated is updated exactly once by the execution of a <bind> construct, after all input resources to that bind have been updated. Thus each resource has a well-defined *old* value, which is its value prior to the start of the execution cycle; and a *new* value, which is its value after the completion of the execution cycle.

When a <bind> construct executes, it has access to both the old and the new values of its input resources. Binds that use new values can be used to model constraint-like updates, for example keeping a data or presentation view in sync with the data being viewed. Binds that use old values can be used to model non-idempotent operations such as adding to a total, appending an item to a list, or inserting a record into a database.

Path expressions play a central role in the Collage language, for example to identify the specific resources whose values are declaratively related by a <bind> construct. A path expressions is a formula for navigating from one resource to one or more other resources via RDF triples. Path expressions in Collage are based on XPath expressions, but Collage generalizes XPath to allow navigation over arbitrary RDF graphs by using the mapping between RDF and XML described above.

End-to-end example

The following example is a simple end-to-end Collage application that provides a form (1) that allows querying and updating a relational database of weather information (2). The <create> construct associates user interface elements such as inputs (3) and triggers (4) with the WEATHERMAN resource class that represents the form. The <let> construct (5) uses the "city" input field to select a row from the database, recording it using the "selected" property. The <bind> construct (6), triggered by the "set" trigger (4), updates the database with the quantity in the "temperature" input field, after converting Fahrenheit to Celsius. A similar <bind> construct (7) retrieves the temperature from the database, converting Celsius to Fahrenheit.



3 <c:create class="c:INPUT" property="temp-input">
 <c:out path="c:label">Temperature:</c:out>
 </c:create>

4 <c:create class="c:TRIGGER" property="set-trigger">
 <c:out path="c:label">Set Temp</c:out>
 </c:create>

5 <c:let property="selected"
 path="/weather-database/row[city=\$city-field]">
 <c:in variable="\$city-field" path="city-field"/>
 </c:let>

7 <c:bind>
 <c:in path="get-trigger"/>
 <c:in variable="\$t" path="selected/temp"/>
 <c:out path="temp-input">{(\$t * 32) * 5 div 9}</c:out>
 </c:bind>

6 <c:bind>
 <c:in path="set-trigger"/>
 <c:in path="temp-input" variable="\$t" passive="true"/>
 <c:out path="selected/temp">{\$t * 9 div 5 + 32}</c:out>
 </c:bind>

As indicated by the dashed boxes above, the application may be distributed among multiple computing nodes, such as a browser and a server. Distributed data structures are formed by triples that connect resources with URLs residing on different computing nodes, such as (5). Distributed execution occurs when <bind> constructs that reference distributed data structures are executed, such as (6) and (7).

Efficient protocols to implement such distributed data and execution models are an active focus of the Collage project. We hypothesize for example that execution of the bind in (6) can be accomplished, at a protocol level, by an appropriate HTTP PUT or POST.

Interaction model

The presentation and user interaction model in Collage allows the description of user interfaces, or application "front-ends", at a range of levels of abstraction. A recursive MVC pattern is supported, allowing the developer to refine an abstract user interface description through successive levels of more concrete specification.

The *model* of an instance of the MVC pattern is represented by a Collage resource. The *view* of MVC is a set of resources associated with the model, whose instantiation is declaratively driven by the <create> construct. The *controller* of MVC is a set of <bind> constructs that update the model resource in response to updates to the view resources, and vice versa.

The set of resources that comprise a view of a model resource may themselves serve as models for other views, thus supporting a recursive MVC pattern. The set of resources comprising a view, together with the <bind>-based controller that connects the view resources with the model resource, may also be seen as a more concrete refinement of an abstraction represented by the model. Conversely, the model may be viewed as an encapsulation of the functionality provided by the view.

Relationship to XForms. Collage builds on and generalizes many of the concepts familiar from XForms to produce a uniform programming model across all application tiers. The XML tree-based MVC design of XForms is made recursive and generalized to RDF graphs in Collage. The concept of view-model and model-model binding is expanded to a general-purpose computation model based on resource-resource binding. Data-driven user interface instantiation is generalized to declarative resource instantiation. The event-driven execution model of XForms is simplified and regularized by the data-centric update-driven execution model of Collage.

Composition model

The declarative data and execution models of Collage support a flexible approach to application composition. We illustrate this here by some examples drawn from application front-ends. However, these programming model capabilities likewise support flexible composition in all tiers of the application.

Flexible decomposition and styling. The choice of a particular view to refine an abstract model resource is determined by the class of the model resource. The Collage/RDF support for multiple and dynamic classification allows a great deal of flexibility in this process because the choice of specific presentation for an abstraction to be made early, by the developer of the abstraction; or later at composition time or run time by a consumer of the abstraction.

For example, an interactive calendar abstraction in Collage may consist of a resource of class DATE whose programmatically computed value is a date to be presented to the user, and whose value is updated as a reflection of the act of choosing a date by the user. A more specific choice of presentation for a DATE model, for example as three fields for year, month and day, may be made by classifying (either early or late) some or all DATE resources in an application with classification DATE3, and defining a set of view resources to be associated with the DATE model resource by virtue of its classification as a DATE3 resource.

every resource of class DATE has a yr, mo, and da resources as its value

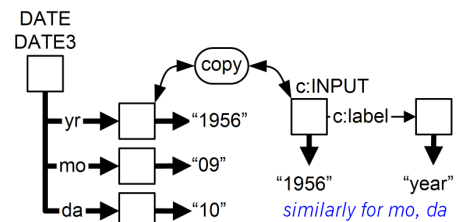
```
<c:with anchor="DATE">
  <c:create-structure>
    <yr/>
    <mo/>
    <da/>
  </c:create-structure>
</c:with>
```

classify every DATE resource as a DATE3 resource as well

```
<c:let anchor="DATE" class="DATE3"/>
```

every DATE3 resource has three associated input fields whose values are bound to the yr, mo, da values

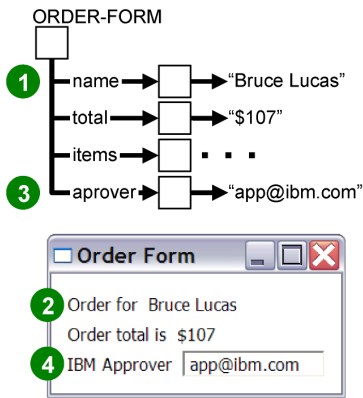
```
<c:with anchor="DATE3">
  <c:create-view class="c:INPUT" ref="yr">
    <c:out path="c:label">year</c:out>
  </c:create-view>
  <c:create-view class="c:INPUT" ref="mo">
    <c:out path="c:label">month</c:out>
  </c:create-view>
  <c:create-view class="c:INPUT" ref="da">
    <c:out path="c:label">day</c:out>
  </c:create-view>
</c:with>
```



Open composition and adaptation. The open approach taken by Collage and RDF to definition of structure and function associated with classes supports flexible multi-organizational composition and adaptation of applications.

For example, suppose that IBM partners with Bookseller to provide IBM employees with supplies such as books. The partnership agreement requires that IBM modify "stock" Bookseller user interfaces and processes, such as adding a provision to specify an IBM approver for each order. This might require that IBM insert an approver field in each submitted IBM order, and a corresponding input field into the ordering page.

The following figure shows a fragment of the Bookseller code for the customer order form, including portions of the definition of the order form model (1) and of the order form presentation (2). Also shown is a fragment of code separately specified by IBM to customize the Bookseller order form, including the addition of an approver field to the model (3) and of a corresponding presentation item (4).



Bookseller Code

```

<c:with anchor="ORDER-FORM">
  1 define structure of model
  <c:create-structure>
    <name/>
    <total/>
    <items/>
  </c:create-structure>
  2 display the customer name
  <c:create-view class="c:OUTPUT" ref="name">
    <c:out path="c:label">Order for</c:out>
  </c:create-view>
  . . .
</c:with>

```

IBM Code

```

<c:with anchor="ORDER-FORM">
  3 define structure of model
  <c:create-structure>
    <approver>app@ibm.com</approver>
  </c:create-structure>
  4 display the approver field
  <c:create-view class="c:OUTPUT" ref="approver">
    <c:out path="c:label">IBM Approver</c:out>
  </c:create-view>
</c:with>

```

Thus while the <with> construct in Collage is comparable in some respects to a class definition (of the ORDER-FORM class in the example above), it is more flexible in that it allows the complete definition of a class to be composed from multiple independently specified sources. This approach supports flexible multi-organization composition of applications.

Device Adaptation. The MVC recursion is grounded in a set of built-in resource classes that may drive existing interaction technologies such as Swing, XHTML, or VoiceXML, representing device-specific atoms of interaction, such as buttons, fields, and voice commands. Just above this most concrete level is a level of resource classes representing more abstract (but still primitive) units of interaction, such as inputs, outputs, and triggers. These primitive units of interaction are in turn collected into larger units, such as pages and flows. Any of these levels of interaction may be connected to back-end data stores and processes.

Collage uses a uniform data and execution model for all levels of this recursive MVC tree, supporting flexible device adaptation. For example device adaptation could be accomplished in Collage using a typical MVC approach of providing for each device a different view on a common model. In Collage however the developer has the additional option of extending the model-view recursion by treating a view for one device (for example a desktop device) as the model to a new view better suited to a different device (for example a mobile device), as illustrated by the following example.

The example shows a simple page for searching titles in a book ordering application. View (1) is oriented toward a desktop display with sufficient space to show details of each book of a list of books all in the same screen. Views (2) and (3) are oriented toward a mobile device. They bind to the view resources of the desktop view as their model. View (2) displays a "master" list of books, while view (3) displays the details for a single selected book. The artifacts created for this adaptation include new view elements (4), binds linking the new view to the old (5) and controlling navigation (6). These are all Collage programming constructs so themselves are available for further adaptation if needed.

