MerlinSDK Adaptation Implementation

This outlines some of the key decisions made, and outstanding issues with the Merlin SDK adaptation implementation as of 5/15/2019.

Interface

Merlin SDK uses a *Fluent Interface*, which is a type of Domain Specific Language (DSL) called an *Internal DSL* [1]. This type of DSL uses constructs within an existing programming language to express a kind of grammar. A notable Internal DSL is JSON, which is a subset of JavaScript. In this paradigm, we focus primarily on the composition of phrases which will create the objects required to construct an adaptation.

The specific implementation that has been selected uses:

- Semantic Model
- Expression Builder
- Method Chaining

In the Merlin SDK Fluent Interface a set of Expression Builders are tasked with populating a Semantic Model with data through Method Chaining. Method Chaining is the ability to chain calls together without having to refer to the "host" object before each call. An Expression Builder acts as a translation layer which translates the DSL into the Semantic Model. This Semantic Model is a functional representation of an adaptation. Separating the language that is used to describe an adaptation from the underlying model which represents it allows both to evolve independently.

What this means is that there are concrete objects such as `Adaptation`, `Activity Type`, and `Resource` which are populated via builders such as `AdaptationBuilder`, `ActivityTypeBuilder`, and `ResourceBuilder`. The interfaces for the two can change over time to suit the needs of Adaptation Engineers via the Expression Builders and Software Engineers via the Semantic Model.

This implementation enables many interesting possibilities. For example:

- The adaptation itself could be treated as a state machine
- A populated Semantic Model can be used to generate code (e.g. JSON)
- A different DSL could be used to populate the Semantic Model (e.g. APGen)

Action Items

In the Merlin SDK, the adaptation schema which serves as the Semantic Model is not a complete model of an adaptation—it's missing both Activity Types *and* Resources. While these two entities are represented as `ResourceBuilder` and `ActivityTypeBuilder` objects in the `AdaptationBuilder`, they are never

reflected in the Semantic Model, which may be causing confusion and difficulty in adding resources of different types.

- Extend the Adaptation schema with resources and activity types.
- Don't expose the underlying Semantic Model to the user (i.e. `Resource`)
- Remove `getResource()` calls as this is the job of the `ResourceBuilder`

Activity Types

The primary reason for deviating from Java class based activity type definitions was the need to support programmatically loading activity types into an adaptation. This is not possible using Java classes to represent activity types. A specific use case is creating activity types from an activity dictionary YAML file.

A second reason was to move away from using reflection to find and parse activity classes. We found that this limited our ability to describe activity types in a simple way, while still making their various properties available. For example, specifying parameters for an activity type within a method would be infeasible because it is (practically) beyond the capabilities of reflection.

Models

In Merlin SDK adaptations, models are separate from activity types. This separation enables the use of different models at runtime. There was a desire to run a simulation with varying fidelities. This can be achieved in several ways:

- Intermix high and low fidelity code within the same model and use some value to determine which path to take
- Use separate models for differing fidelities and specify which fidelity to use at runtime

Using separate models to represent different fidelities is a more maintainable solution. In addition to the ability to specify the fidelity at runtime, it opens up the potential to simulating different parts of a schedule with different fidelities.

Action items

The current version of the Merlin SDK does not implement fidelity switching. A future task could add this very simply with an interface similar to this (example only):

```
ActivityTypeBuilder biteBanana = adaptation.createActivityType()
    .withName("BiteBanana")
    .withHighFidelityModel(new HighFidelityFruitModel())
    .withLowFidelityModel(new LowFidelityFruitModel())
    .withCustomFidelityModel("myFidelity", new FruitModel());
```

Here, a high and low fidelity model are registered. A custom fidelity model would allow the user to specify the name of their fidelity. This could be a more

flexible, but probably unnecessary way to add different fidelities. At runtime a value would be passed with each activity instance which specifies which level of fidelity to use. This could also be specified for an entire schedule.

Another issue with the current implementation of models is that they should not take an instance of the model, rather a class type should be passed, and the adaptation runtime service should be responsible for instantiating models when they are needed. For example:

```
ActivityTypeBuilder biteBanana = adaptation.createActivityType()
    .withName("BiteBanana")
    .withModel(FruitModel.class);
```

Resources

One aspect of Expression Builders to consider, is when to invoke a new one. Fowler suggests [2] representing complex tree structures with distinct Expression Builders using the Composite pattern [3]. Currently, most functionality comes from the `AdaptationBuilder`, but `ActionTypeBuilder`, `ResourceBuilder`, and `ParameterBuilder` objects are invoked separately. This mostly fits within Fowler's recommendation as both action types and resources can be hierarchical. Parameters may need to be refactored so as to better conform to this paradigm.

There are several ways to implement differing types of resources. One approach is to subclass a resource, however this should only be done if the Resource itself is significantly different from other resources. In the case of the `LinearCombinationResource`, the resource itself is no different from the other resources, thus a subclass is inappropriate. It would be better to define a strategy for representing the type of relationship between resources, which ultimately defines an algorithm for producing a new value in a parent resource. The Strategy pattern [4] addresses this use case.

Children of a resource that is using the linear combination strategy may need to have a multiplier applied to their value when they are integrated into the parent resource. Resources can be reused in other linear combinations and the multiplier may be unique to each relationship. One way to address this requirement in a general-purpose way, is to include relationship-specific metadata, which would be stored on the parent resource.

Action Items

- Child `ResourceBuilder` objects should be invoked separately and stored within local variables to be added later.
- Implement the Composite patter on resources by creating an `addResource` interface on the `ResourceBuilder` which would be used to add child `ResourceBuilder` objects.
- Implement the strategy pattern interfaces.

- Define the resource strategy on `ResourceBuilder` and implement it on the top level resource.
- Add a `strategyType` field to the `Resource` class which will be set by the builder and read at runtime.
- Create a `LinearCombinationResourceStrategy` which implements the logic for associating resources at runtime.
- Add a `MetadataBuilder` with a method for setting the multiplier
- Specify metadata when the child resource is added to the parent resource
- Incorporate the associated values into the strategy at runtime

References

1. M. Fowler and R. Parsons, "Implementing an Internal DSL," in *Domain-specific languages*. Boston, MA: Addison-Wesley, 2011, ch 4, pp. 67.
2. M. Fowler and R. Parsons, "Expression Builder," in *Domain-specific languages*. Boston, MA: Addison-Wesley, 2011, ch 32, pp. 343.
3. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, "Structural Patterns," *Design patterns: elements of reusable object-oriented software*. New Dehli: Pearson Education, 1994, ch 4, pp. 163.
4. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, "Behavioral Patterns," *Design patterns: elements of reusable object-oriented software*. New Dehli: Pearson Education, 1994, ch 5, pp. 315.