**DaySim Developer’s Guide  
Partial draft  
October 11, 2012**

**UTILITY AND SIZE FUNCTIONS IN CHOICE MODELS**

The following entries explain the methods that can be used in DaySim model classes to specify utility and size functions. They include the capability of specifying ‘components’ that are shared by multiple alternatives.

**ChoiceModelHelper**

--performs generic functions needed by all models to do their job

**ChoiceProbabilityCalculator**

--performs model-specific functions needed by all models to do their job

--includes the **key** of the **case** for which the model is being used, such as person.Id  or tour.Id.

choiceProbabilityCalculator = \_helper.GetChoiceProbabilityCalculator(int key)

--creates an instance of **ChoiceProbabilityCalculator** for the **case** identified uniquely by **key**

--**\_helper** is an instance of **ChoiceModelHelper**

alternative = choiceProbabilityCalculator.GetAlternative(int index, bool available, bool isChosenAlternative)  [we should make it so that we can refer to choiceProbabilityCalculator.alternative(index) after it has been added]

--creates or updates an **alternative** for the **case** being handled by **choiceProbabilityCalculator**

--**index** is the **alternative** number

--**available** determines whether **alternative** is available

--in estimation mode, **isChosenAlternative** determines whether **alternative** was chosen

choiceProbabilityCalculator.CreateUtilityComponent(int index)

--creates a new component to be added to the utility functions of one or more of choiceProbabilityCalculator’s alternatives

--**index** uniquely identifies the utility component within the model

--within the model, each CreateUtilityComponent statement must apply to ALL cases.

choiceProbabilityCalculator.CreateSizeComponent(int index)

--creates a new component to be added to the size functions of one or more of choiceProbabilityCalculator’s alternatives

--**index** uniquely identifies the size component within the model

--within the model, each CreateSizeComponent statement must apply to ALL cases.

component = choiceProbabilityCalculator.GetUtillityComponent(int index)  [we should make it so that we can refer to choiceProbabilityCalculator.component(index) after it has been added]

--retrieves a uniquely **indexed utility component** from instance **choiceProbabilityCalculator** of ChoiceProbabilityCalculator

component = choiceProbabilityCalculator.GetSizeComponent(int index)  [we should make it so that we can refer to choiceProbabilityCalculator.component(index) after it has been added]

--retrieves a uniquely **indexed size component** from instance **choiceProbabilityCalculator** of ChoiceProbabilityCalculator

component.AddUtilityTerm(int parameter, double value)

alternative.AddUtilityTerm(int parameter, double value)

 --adds an additive utility term to **component,** or to **alternative**, consisting of **parameter** \* **value**

--**alternative** or **component** belong to an instance of ChoiceProbabilityCalculator

--**parameter** is a model coefficient

--**value** is a model variable (or may be a constant, such as 1.0)

--for **component**s, it is assumed that any parameter in a term added to a **size component** is a size variable parameter, and that any parameter in a term added to a **utility component** is NOT a size variable parameter.  (There are no checks on this right now, so it's up to the user to make sure this is the case)

--any number of utility terms can be added to **component** or **alternative** by repeated use of AddUtilityTerm

--within a model, a particular AddUtilityTerm statement can be applied conditionally to certain cases and not to others

alternative.AddUtilityComponent(Component component, double factor)

--adds all the terms of an existing **utility component** to an **alternative**’s utility function after multiplying each of its terms by **factor**

--**alternative** is an alternative within the instance choiceProbabilityCalculator of ChoiceProbabilityCalculator

--within the model, a **component** can be added to all or some of the **alternative**s, and can be added more than once to a single **alternative**

--within the model, each AddUtilityComponent statement must apply to ALL cases

--the value of **factor** must be the same for ALL cases, but can vary across **alternative**s)

alternative.AddSizeComponent(Component component, double factor)

--adds all the terms of an existing **size component** to an **alternative**’s size function after multiplying each of its terms by **factor**

--**alternative** is an alternative within the instance choiceProbabilityCalculator of ChoiceProbabilityCalculator

--within the model, a **component** can be added to all or some of the **alternative**s, and can be added more than once to a single **alternative**

--within the model, each AddSizeComponent statement must apply to ALL cases

--the value of **factor** must be the same for ALL cases, but can vary across **alternative**s)

**CLIENT-SPECIFIC HOUSEHOLD INPUT DATA**

DaySim uses default data types with associated interface classes and wrappers (e.g. Household, IHousehold and HouseholdWrapper) for the various household-related data classes in the DomainModels directory of the DaySim solution. The basic data type identifies the data that is read from the input data, the interface defines what must be in a Household, and the wrapper adds derived properties and methods. These three classes exist for several data types, including Household, HouseholdDay, Person, PersonDay, Tour and Trip; and the basic data type and wrapper exist for JointTour, PartialHalfTour and FullHalfTour.

In addition, Factory and WrapperCreator classes exist for Household, HouseholdDay, Person, PersonDay, Tour and Trip. This makes it possible to read additional client-specific data items for these six data types, by taking the following steps (the Actum client-specific classes provide an example to emulate):

1. Create client-specific classes for all the (six) data types for which Factory classes exist.
   1. For the client-specific basic data type, clone the default class and then add the client-specific data elements. If there are no client-specific data items for a particular data type, then a client-specific basic data type is not needed (although DaySim still needs a client-specific wrapper and wrapper creator for all six factories, as follows).
   2. For the wrapper, inherit the data type’s default wrapper and add the client-specific input data properties and additional desired properties and methods.
   3. Create WrapperCreator classes using Actum WrapperCreators as examples.
2. Modify Engine.cs, the configuration file, and DaySimModule.cs to support the new classes
   1. Add lines to Engine.InitializeFactories() patterned after the Actum examples. Each one uses the client-specific literal of your choice to identify the lines associated with the client-specific data (e.g. “Actum”)
   2. Add a parameter to the configuration file that identifies the client-specific literal used in the Engine statements for the client (e.g. DataType=”Actum”)
   3. Add lines to DaySimModule.Load() pattenred after the Actum examples.
3. Supply the input data in the same format as the default data inputs, but with the client-specific data items added to the right end of the record.
4. When using the client-specific properties, it is necessary to cast to the client-specific class.

If a change is made to a default basic data type, such as adding a new standard input data item to the Household class, then the change must be made to the default basic data type, its associated interface, and all the corresponding client-specific data types. The client-specific basic classes must be updated because they do not inherit from the default basic data type). However, if a change is made to the default wrapper of a data type, then the client-specific wrappers do NOT need to be updated because they inherit from the default wrapper class.

**CUSTOM SAMPLING WEIGHTS SETTINGS FOR DESTINATION SAMPLING**

DaySim uses a C# class that supports its destination sampling methods. This class determines the number of destination sampling segments, sets the size and impedance parameters for destination sampling, and provides the logic for selecting the correct segment in a given sampling situation. The default class is named “SamplingWeightsSettings”. In addition, a SamplingWeightsSettingsFactory exists that makes it possible to use non-default classes that use different segments, sizes, weights and segment selection logic. This is accomplished through the following steps:

1. Create the desired non-default class. It is easiest to do this by cloning and modifying the default class or an existing non-default class.
2. Add a line to Engine.InitializeFactories() to support the new class, patterned after the existing lines for SamplingWeightsSettings.
3. Add a line to the configuration xml file that identifies the class name that should be used for sampling weights settings. For example, SamplingWeightsSettingsType="SamplingWeightsSettingsSimple" instructs DaySim to use SamplingWeightsSettingsSimple.cs, which implements a simplified sampling scheme in cases where memory footprint is an issue.