Мэйкер. Действия, методы и свойства

1. Постановка задачи

Каждый список содержит элементы, которые могут быть обработаны одной моделью мейкера. Однако, когда образуются связанные списки, то следует ожидать, что в зависимости от характера необходимости индивидуального отображения, как списка так и самих списков, исходная модель мейкера должна быть модифицирована. В общем случае мейкер (**maker**) это совокупность действий (**actions**), методов (**methods**) и свойств. (**properties**)

Следует ожидать, что имеется и соответствующий список, в котором должны находиться отдельно списки действий, методов и свойств. Ясно, что у элементов типа «**maker**» имеется свой мейкер создания шаблонов и соответственно должен быть и список шаблонов «**maker**».

Следует понимать, что имеется задача «поднять себя дергая собственные шнурки».

В чем заключается задача. Надо для списка (неважно какого) выделить элемент, который является мейкером элементов. Найти этот элемент и загрузить из этого элемента модель мейкера для данного списка и элементов этого списка. По всей видимости это будет делать удобно через имя мейкера, что предполагает иметь механизм защиты мейкеров от использования имен или механизм назначения имен мейкера предусматривающий назначения такого имени юзера (исключения подлога). К сожалению, такого механизма в концепции **Cotonti** нет. Вводить блокировки вовсе необязательно, достаточно «раздвинуть» имя такого элемента, допустим добавить в конец имени элемента символы «_», добавить столько , пока не будет конфликта имен. Аналогично необходимо будет проделать аналогичную операцию и действиями, методами и свойствами. Соответственно будут и списки шаблонов, чтобы можно было бы создавать мейкеры,

Итак, имя списка порождает имя мейкера списка. Следовательно, достаточно иметь в списке мейкеров элемент с таким именем. Этот элемент, можно рассматривать как исходную модель. В этом элементе будут иметься связанные списки элементов с действиями для элементов, списки свойств элементов и методов работы со значениями по свойствам. Поэтому, когда мы создаем список товаров, то уже будем знать, что необходим элемент в списке мейкера с именем «**товары**». Очевидно, что будут отдельные группы описывающие мейкер для списка и мейкер для элементов. Теперь переходим к самому списку, то замечаем, что и у списка имеется имя, следовательно, необходим элемент, который как бы совпадает с типом элементов, но это не простой элемент, этот элемент описывает особенности мейкера для данного списка, и, следовательно он должен храниться в списке элементов подпадающих под общий мейкер. Будем считать, что это элемент типа «мейкер» поле **page_state** таблицы **sed_pages.** Поэтому, когда будем создавать новый элемент в списке, то можно будет указать, что это элемент типа «мейкер». Очевидно, у этого элемента будут иметься аналогичные связанные списки акций, методов и свойств, которые в совокупности с базовой моделью элементов подпадающие под мейкер, будут образовывать уже конкретную модель для конкретного списка элементов. Например, имеется список «сопутствующих товаров» то в списке сопутствующий товаров с кодом «**Ac_goods**». Далее имеются еще и связанные списки, то для каждого связанного списка можно создать такой же элемент, но уже с именем состоящим из кода элемента и кода связанного списка, то есть будет элемент «**КодСписка_КодЭлемента**», который будет уточнять модель мейкера для связанных списков элемента. Переходя к варианту мейкера, можно обнаружить, что и тут требуется такой элемент, очевидно, что такой элемент будет типа мейкера и образовывать свою уникальную идентификацию. Переходя, на аналогию языка PHP можно смело назвать этот элемент «**this**», однако в системе может быть только одно уникальное имя для элемента, пусть это имя будет «**this_КодЭлемента**».

Легко посчитать, что количество элементов в системе становиться по крайней мере в два раза больше. К тому же непонятно, а как быть со значениями. Ясно, что «тащить» свое

«зеркало»- уточненную модель должен сам элемент. Для этого можно воспользоваться страницами элемента, но такие страницы должны носить приватный характер и не иметь доступа для редактирования юзером. Юзер может добавлять или удалять, но юзер не должен иметь права редактировать, иначе будет «взлом» системы. То есть такие страницы должны быть типа «**protected**»

2. Свойства

Для описание свойств элемента необходим метод, который позволяет организовать отображение, ввод и редактирование свойств элемента.

Что такое свойство? Свойство это отображение элемента в некотором множестве описывающих совокупность элементов, к которым применимы индивидуальная совокупность операций над этими элементами. Поэтому, под свойством в системе **концепции MobiCot** понимается отображенное множество. Элементы, в таком отображенном множестве считаются значениями элемента по свойству. Множество, как известно, это совокупность элементов объединенных общим признаком. Любое множество содержит пустой элемент. Поэтому в списках (множествах) всегда содержится некоторый элемент, который позволяет по его подобию создавать для элемента - конкретное значение по указанному свойству элемента.

2.1. Проект реализации

Как понимать свойство. Допустим у нас имеется элемент «**Апельсин**». Если мы запишем свойство в такой нотации (язык ИИ «**Проза**»)::

Цвет:Апельсин=”Оранжевый”;

то становиться ясно, что у апельсина есть свойство «**Цвет**», а значение элемента «**Апельсин**» по этому свойству будет равно **«Оранжевый»**. Следовательно, если в нулевом элементе хранить структуру элемента, а в самом элементе хранить значения по этому свойству, то можно получить для нулевого элемента общую структуру для всех свойств элементов в списке, а в конкретном элементе значения этих элементов. С другой стороны если переходить к нотации «**Прозы**», можно написать следующее выражение:

Язык”ru”:Наименование:Апельсин=”Апельсин”;;

Что обозначает, что в множестве Язык имеется подмножество «**ru**», в котором элемент «**Апельсин=**» по свойству «Наименование» имеет значение **”Апельсин”**.

Теперь, попробуем отобразить полученную нотацию средствами языка **PHP** и системы **Cotonti**.

У нас возникает задача определить свойства, но не просто свойства, а свойства для определенного мейкера, следовательно элемент описывающий свойства должен находится в общем для элементов списков с данным мейкером. Но конкретное свойство безотносительно данного мейкера, следовательно необходим связанный список свойств, который раскрывает данный список. С другой стороны сами элементы, хотя имеют общий мейкер, могут быть созданы как варианты представления этого мейкера. Создание нового элемента в системе **концепции MobiCot** производится копированием заранее разработанного элемента-шаблона. Поэтому, если у определенного шаблона будет

связанный список элементов, наборов свойств, то можно будет разработать операцию, которая будет собирать эти отдельные свойства вместе и копировать их в отдельной странице свойств шаблона. А когда шаблон будет использоваться по назначению, на создание нового элемента, то наряду с дублированием описательной части, можно будет создать общедоступный элемент с этими свойствами, у которого, отдельная страница будет соответствовать наименованию шаблона.

Итак, вначале создадим категорию **methods(**методы**)**, а в ней подкатегории (списки) **-properties(**новые свойства**)** и **-actions(**новые действия**),** Затем добавим две подкатегории: **properties(**Свойства**)** и **actions(Д**ействия**):**

Код	Путь	TPL	Заголовок	Группа
methods	10	-	Методы	☑
-properties	10.1	-	Новые свойства	☐
-actions	10.2	-	Новые действия	☐
properties	10.3	+	Свойства	☐
actions	10.4	+	Действия	☐

В списке **properties(**Свойства**)** расположим элементы описывающие, например его основные свойства: **price(**цена**), price_supplier(**цена поставщика**), quantity(**количество**)**. А такие свойства как **color(**цвет**), weight(**вес**), length(**длина**)** будем считать специфическими, то есть такими свойствами, которые будут просто находиться в списке свойств, но такие, которые юзер, владелец элемента сможет добавить как дополнительные свойства.

Для того, чтобы создавать новые свойства, необходим элемент элемент-шаблон. Ясно, что в описательной части необходимо писать, что такое свойство. Для описания свойств воспользуемся методом представления элементов **JSON** для которого в **PHP** имеются две функции:

json_decode — Декодирование строки **JSON**

json_encode — Кодирование строки **JSON**

Следовательно, если описать в терминах **JSON** свойства**,** то получится универсальный метод описания свойств и (или) операций.

Введем соглашение, что элемент это некоторый массив элементов свойств со своими значениями. Также нам известен метод кодирования частей элемента в системе **концепции MobiCot.** Для выборки части элемента (то есть, если представить элемент множеством, то его части это элементы этого подмножества):

кодирование элемента опция **al=ассоциативная метка элемента**;

кодирование части элемента опция **pag=ассоциативная метка элемента**.

Ассоциативная метка может кодироваться:

метка1=значение метки1/..... /меткаN=значение меткиN

То есть, если в опциях элемента будет записано:

pg=properties=ru

то это будет обозначать, что необходимо взять в элементе его элемент **properties**, в котором есть массив описывающий на русском языке свойства элемента. Но если будет записано без значения :

pg=properties

то это будет обозначать, что необходимо взять значение в текущей языковой настройке.

Однако можно заметить, что таких похожих частей элементов для мейкера в одном списке будет множество, что, в конечном итоге приводит к излишнему размеру таблиц. Однако для описания конкретного свойства этот механизм подходит. Опишем часть элемента

[newpage:properties=ru][title]Описание свойства price(цена)[/title]

```
{
"price":
      {"name":
            {
            "ru":"Цена","en":"Price"
            },
      "value":
            {
            "rub":10,"usd":0.33
            }
      },
"rub":
      {"name":
            {
            "ru":"Руб.","en":"Rouble"
            }
      }
}
```

В описании свойств присутствуют два свойства, это свойство **price (Цена)** и свойство **rub(Руб.)**. Первое свойство **price (Цена)** относится к категории специфических, тогда как свойство **rub(Руб.)** можно отнести к общедоступным (или общеупотребительным). Однако, если присмотреться к свойствам, то можно заметить, что свойства, в конечном виде сводятся к значением свойств.

Обращаясь к языку ИИ «**Проза**» необходимо привести нотацию действий в терминах этого языка:

Обращение!действие:элементы;

Под **Обращением** понималось то устройство, тот программный компонент или нечто самостоятельное, которое способно выполнить указанное **действие**, например, для рассматриваемого примера, этот будет записано так:

Мейкер «Товар»!отобразить и ввести:элемент:код=«Пескарь»;;

Как видно, требуется обращение. И не просто обращение, а обращение **Мейкер «Товар».** В общем случае, найти подходящий мейкер это задача среды, в которой происходит движение элементов. Нам важны действия **отобразить и ввести**. **Отобразить и ввести,** это специфические операции над элементами среды приложения мейкера. В системе **концепции MobiCot** среда приложения это список, для которого уже известен ее мейкер. А

вот откуда взять конкретный мейкер и как им воспользоваться служит часть «Действия(**actions**)»