Viking direct marketing system

OOPSLA '00 DesignFest problem, October 15, 2000

Design team:

Bill Hofmann - moderator
Dennis Mancl - recorder
Aaron Mohrman
Mark Steffel
Igor Vinnychuk
Michael Welch

From the problem statement:

The Viking is a system that maintains information about customers and can send them letters tailored with important, relevant, and time-critical information. The Viking is the core of a direct marketing system: it will directly interact with users to configure the campaigns and will work with other system components to accomplish the mailing itself.

Class diagram for Viking

The central classes in this system are:

- **Customer** (holds the customer's name, knows which Letters have been generated that will be sent to the customer, holds a set of Plug_in objects that will be used for expanding a Template)
• **Template** (holds the text that will be used to generate multiple Letters, knows how to get information out of Tag objects when it is expanded)
• **Tag** (holds a key name and a default value for that key, can navigate to the Plug_in value that will be used [for Simple Tag] or to the database [for Order Tag])
• **Customer Selector** (gets a customer list on request -- each Customer Selector subclass has a different mechanism for specifying the customer list)
• **Letter** (can get the text that needs to be printed or emailed on demand -- it might be locally cached, or it might be generated whenever requested)
• **Mailing address** (knows a specific email address or postal address that will be used to send Letters to one or more Customer)
• **Mailing system** (has the control logic to "process a Letter" and knows how to send the Letter using one of the Mailing address objects for the addressing operation)
• **Delivery attempt** (keeps track of the status of a Letter object -- it might be "generated", "in transit", "successfully delivered", or "failed delivery"; also holds a tracking string [key] that might be found in returned email messages)
• There will also be a **Template Group** class (see diagram below for "User story 6 -- supplement to object model" for details)

**Risks (possible development problems and design evolution problems)**

1. Need to add new "Mailing address" subclass when a new "Mailing system" class is added to the system [user story 3]
2. "Tracking string" may be an inadequate identifier (not unique, inaccessible on failure, etc.) [user story 8]
3. Users need to have flexible "Customer selector" classes, but users might lack experience to write their own [user story 4]
4. Developers may decide to go too far and create their own programming language for "Customer selector" [user story 4]
5. Template Group objects will be complex to update [user story 6]

**Sequence diagrams for selected user stories**

In user story 1, the system must be able to generate a Letter (by expanding a Template correctly for a particular Customer).
In user story 3, the system must be able to mail Letters to a group of Customers. The Mailing system must be flexible, so the mail can be sent by email or by "paper mail".

**User story 3**

```
Controller --|----------|
             |          |
Mailing     |Letter    |
system      |Customer  |
             |Delivery  |
             |attempt   |
```

```
send_letters(Letters_list, Mailing system)
```

```
Do for each letter in the Letters_list
```

```
get_customer;
get_address(Mailing system)
send(Letter, Mailing address)
```

```
create
set_status("sent")
```

In user story 4, the group of Customers that will have letters generated for them must be selected by some kind of simple criterion. The Customer selector will encapsulate the criterion.

**User story 4**

```
Controller --|----------|
             |          |
Customer    |Template  |
selector     |          |
             |
```

```
check_letters(Letters_list, Customer selector)
```

```
get_customer_list
```

```
Do for all customers in the list
```

```
generate_a_letter(Customer)
```

```
... see user story 1...
```

In user story 8, the system must record information about Letters that have failed delivery. In this design, the corresponding Delivery attempt object will hold that status information.

**User story 8**

```
Controller --|----------|
             |          |
Delivery    |Delivery  |
attempt      |attempt   |
finder      |          |
```

```
post_delivery_details(Delivery Attempt)
```

```
find(Delivery Attempt)
```

```
update_status("failed")
```

In user story 7, the system must handle Tag objects that can produce a list of values. The values might be retrieved from a database, or they might be determined by following associations in the object model. This scenario illustrates the scenario "find a list of companion products from all the Orders placed by the
In user story 6, the Template Group class is introduced to allow customized Letters to be sent to different kinds of Customers -- for example, a Template Group might contain Templates in different languages. Each Template in a Template Group has a priority, so the Templates will be applied in priority order, and duplicate Customers will not be sent multiple Letters.

User story 6 -- supplement to object model

Last modified: Oct. 21, 2000