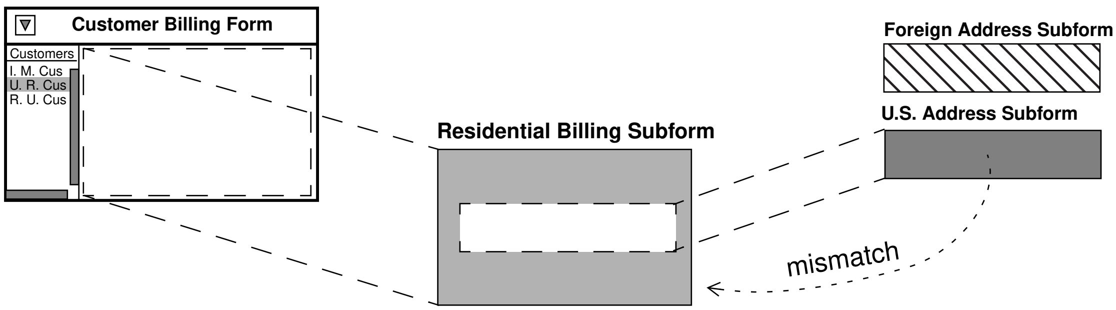
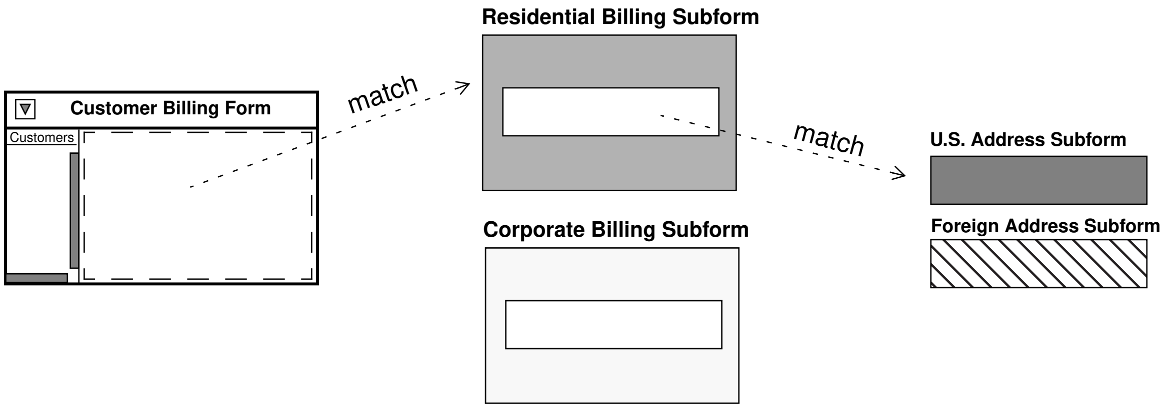
Example: The following sketch depicts an application where the window displays a customer billing form. In this example, customer “U. R. Cus” (selected in the parent window list widget) is a residential customer with a U.S. address. When the user changes the country for this customer to a foreign address the “child” **U.S. Address Subform** sends #mismatch to the “parent” **Residential Billing Subform**. At this point, the **Residential Billing Subform** will poll for the new matching subform using Subform Selection.



The following Smalltalk code implements the test and notify behavior for the **U.S. Address Subform**. The #stateChange message tests for a match, and in the event of a mismatch, notifies the parent form by sending a #mismatch message. The #mismatch message of **Residential Billing Subform** simply sends #selectSubform to itself to select the appropriate subform.

```
USAddressSubform>>stateChange
    self countryIsUS
        ifFalse: [self parent mismatch]

ResidentialBillingSubform>>mismatch
    self selectSubform
```



The following Smalltalk code implements the test and notify behavior for the **Residential Billing Subform** subform. The #isResidential message validates a match. If a match is detected, the state data (in this example aCustomer) is stored by invoking the #setSubject: message and true is returned to the parent. Otherwise false is returned indicating that "no match" was found.

```
ResidentialBillingSubform>>match: aCustomer
    (aCustomer isResidential)
        ifTrue:
            [self setSubject: aCustomer.
            ^true]
        ifFalse: [^false]
```

7.0 Pattern: Subform Mismatch

Problem: How does the "parent" subform determine when a "child" subform becomes inactive based on a change of state data in the "child" subform?

Context: A GUI based application where subforms are selected using Subform Match and key application state data resides with the "child" subform.

Forces: Key application state data for a "child" subform can change as data is entered/edited by the user.

Polling the subforms for applicability wastes CPU resources and potentially allows ambiguous conditions to exist in the application for the duration of the polling interval.

Centralized subform de-selection can be difficult to develop and maintain particularly when there are numerous forms with complex selection criteria. Selection can also become complex due to the intricacies of application data and the relationship of application data to subform dependencies.

Solution: Have the subform notify its parent when it no longer matches the selection criteria, i.e., when there is a "mismatch". This is easily accomplished by retesting matching criteria each time there is any change in key application state data.

Resulting Context: De-selection is decentralized and distributed to the subform that understands its state data requirements. Decentralizing simplifies and reduces the impact to the application for adding and updating subforms. Since "parent" subforms hold onto their "child" subforms it facilitates rapid search and further reduces complexity.

```
                    subForms do:
                        [:each | |subformWidget |
                        subformWidget := (self builder componentAt: each name) widget.
                        subformWidget client: each spec: each windowSpec].
                subforms]
            ifFalse: [subforms]
```

The following Smalltalk code implements the polling behavior for “parent” subforms. First, the #selectSubform message makes all “child” subforms invisible by sending #beInvisible to them. Then it searches through its collection of prebuilt subforms looking to detect the matching subform. The subform that is selected is made visible by sending #beVisible to it.

```
CustomerBillingForm>>selectSubform

    self subforms do: [:each | (self builder componentAt: each name) beInvisible].
     (self subforms detect: [:each | each match: self selectedCustomer]) beVisible
```

6.0 Pattern: Subform Match

Problem: How does the “parent” subform select the appropriate “child” subform based on state data?

Context: A GUI based application where Subform Selection is used.

Forces: The number of subforms for a given application may be dynamic.

Subforms which are composed of a dynamic number of “child” subforms will need to be selected at runtime.

Centralized subform selection can be difficult to develop and maintain particularly when there are numerous forms with complex selection criteria. Selection can also become very complex due to the intricacies of application data and the relationship of application data to subform dependencies.

Solution: Every time there is a change in key application state data, poll the collection (cache) of subforms and let them identify themselves as a “match” for the state data. This is accomplished by associating a routine (method) with a subform that is able to test the key application state data for matching criteria.

Resulting Context: Selection is decentralized and distributed to the subforms that understand their state data requirements. Decentralizing simplifies and reduces the impact to the application for adding and updating subforms and associated “child” subforms. Having “parent” subforms hold onto their “child” subforms facilitates rapid search and further reduces complexity.

Subforms can be dynamically added or removed from the system without changing the selection criteria for existing (or remaining) subforms.

Example: The following sketch depicts an application where the window displays a customer billing form. In this example, the customer selected in the parent window is residential with a U.S. address. The responsibility for determining if a subform is a match is relegated to the subform itself.

5.0 Pattern: Subform Selection

Problem: How do you manage a collection of subforms that become active/inactive based on state data?

Context: A GUI based application using Alternative Subforms where a change in subform state data results in a change in subform selection.

Forces: Static assignment of subforms could result in a proliferation of dedicated, specialized and therefore, un-reusable forms.

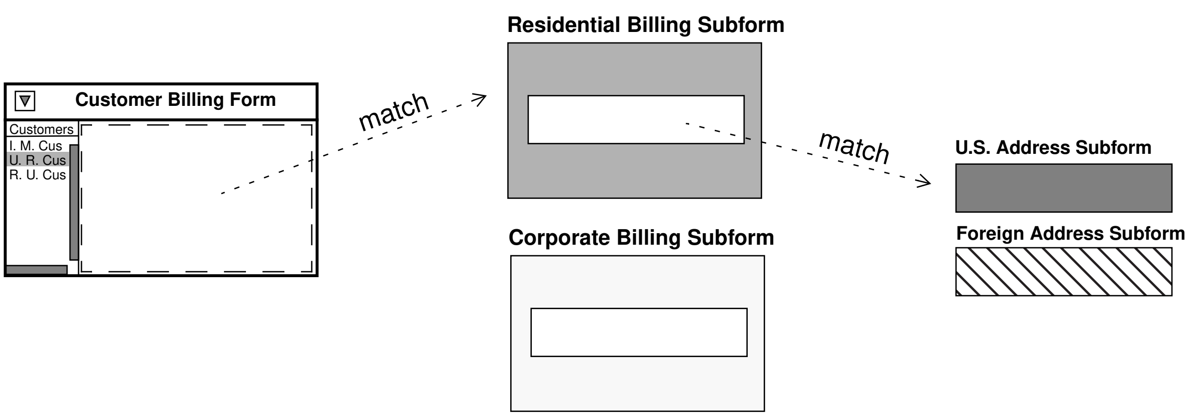
Forms need to conform rapidly to data as it is entered or changed.

Subforms are typically selected according to the value (or values) of key application state data.

Solution: Build each subform when the application is started and cache them for later use. When there is a change in key application state data, and the "parent" subform owns/maintains it, then the "parent" polls each "child" subform using Subform Match. If the key application state data is owned/maintained by the "child" subform, then the "child" subform uses Subform Mismatch to notify the "parent" that it needs to start the polling process to find a new matching subform; at which time the "parent" will again use Subform Match to find the new matching subform.

Resulting Context: Caching the subforms facilitates the dynamic selection of subforms at runtime. Subform Selection speeds up the application because switching between smaller "child" subforms is inherently less costly than switching between entire forms. The most used forms may be ordered at the front of the cache to facilitate rapid search.

Example: The following sketch depicts an application where the window displays a customer billing form. In this example, the customer selected in the parent window is residential with a U.S. address. The window polls its collection of subforms (**Residential Billing Subform** and **Corporate Billing Subform**), and based on the current state data, the **Residential Billing Subform** is found to be a match. The **Residential Billing Subform** in-turn polls its collection of address subforms (**U.S. Address Subform** and **Foreign Address Subform**) to select the appropriate "child" address subform, i.e., the **U.S. Address Subform**.



The following Smalltalk code creates the cache of application subforms.

```
CustomerBillingForm>>subForms

    ^subForms isNil
        ifTrue:
            [subForms := List new.
            subForms add: (ResidentialBillingSubform newForParent: self).
            subForms add: (CorporateBillingSubform newForParent: self).
```

Designing each form from a basic set of widgets is redundant, error prone, and often leads to subtle differences between forms for the presentation of the same information.

Solution: Create a subform that can be reused on multiple forms. When a form needs to be dynamically altered at runtime use Alternative Subforms.

Resulting Context: Entire portions of a form can be reused without the cost associated with rediscovery and redesign. Subform behavior can be overridden allowing alteration of basic widget behavior, e.g., subform initialization can be different for two different forms to allow read/write capability on one form and read-only capability for the other.

Example: Reusable Subform pattern can be employed to display/edit a customer's address that may appear on multiple parent forms, e.g., customer billing and customer ordering forms.

4.0 Pattern: Alternative Subforms

Problem: How do you dynamically change part of a form at runtime based on state data?

Context: A GUI based application using Subform and/or Reusable Subform where the appearance of the form needs to change as data is entered by the user.

Forces: People are better at recognition than recall, therefore presenting a consistent data layout is essential for the user to understand and use an application interface.

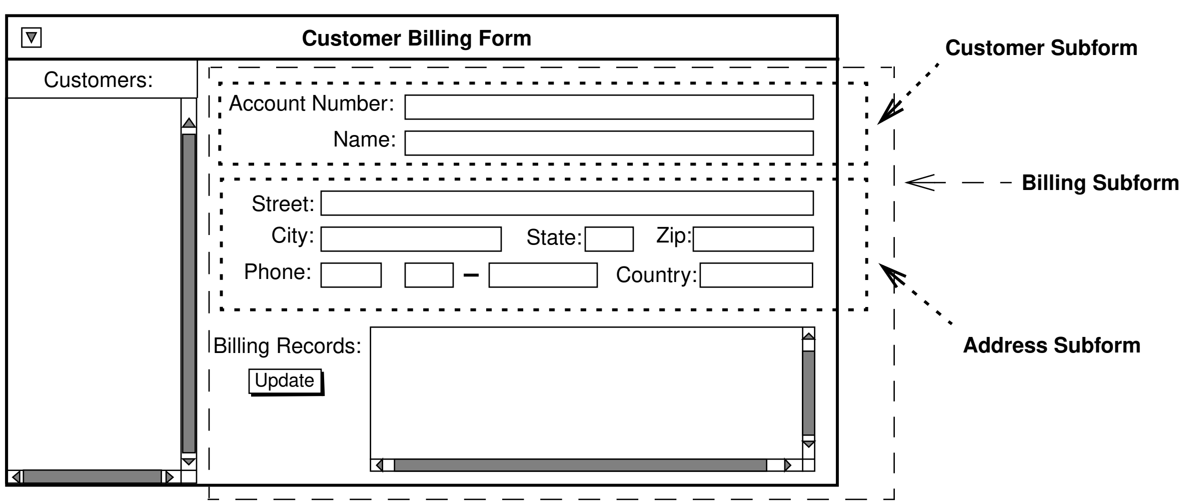
Creating multiple forms and switching between them is costly in terms of development and runtime.

Subtle variations in the data could result in a proliferation of forms that differ by a few widgets.

Solution: Create one subform for each variation of the widgets that change and alternately select the appropriate subform at runtime. Use Subform Selection to manage the selection of these subforms.

Resulting Context: The form will have the appearance of multiple forms without the cost of creating and maintaining separate forms.

Example: The Alternative Subforms pattern is extremely useful for varying the form based on data content. For example, as depicted below, the **Address Subform** could be different for customers from different countries, e.g., changing the Country field could result in an entirely different address subform being selected.



2.0 Pattern: Subform

Problem: How do you compose forms to facilitate ease of design and development?

Context: A GUI based application where forms are used.

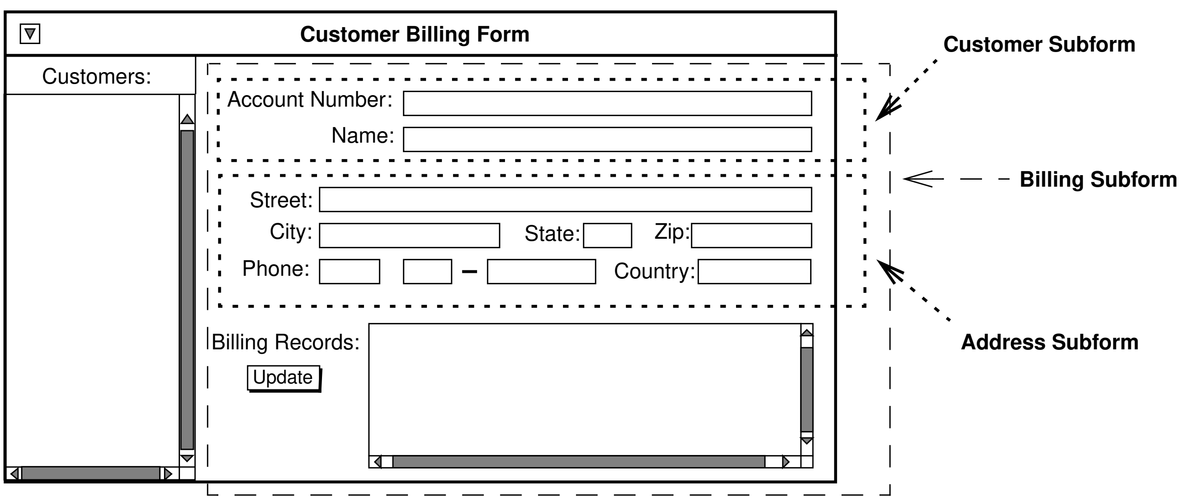
Forces: Forms decompose into categories of related data.

Complex forms (i.e., forms with a significant number of widgets) often have a significant amount of associated behavior that must be organized, designed and maintained.

Solution: Build a single window and employ subcanvases to contain the widgets for all or part of a form where there is a one-to-one correspondence between a form and a subcanvas. Subform can be recursively applied by creating "child" subforms to hold onto a subset of a "parent" subform's widgets.

Resulting Context: Subform facilitates the encapsulation of behavior and associated widgets. It makes it easier to reuse portions of an interface on multiple forms (see Reusable Subform). Subforms can also be used dynamically to alter the appearance of a form at runtime (see Alternative Subforms).

Example: The following sketch illustrates the use of a "parent" **Billing Subform** composed of two widgets (a list and an action button) and two "child" subforms: **Customer Subform** and **Address Subform**. The **Customer Subform** contains the widgets that identify a customer by name and account number. **Address Subform** contains widget's for a customer's address.



3.0 Pattern: Reusable Subform

Problem: How do you minimize development and maintenance effort when collections (subsets) of the same widgets appear on multiple forms?

Context: A GUI based application where Subform is used.

Forces: Designing application forms is time consuming, subjective, and iterative requiring multiple prototype reviews with a number of prospective users.

People are better at recognition than recall, therefore presenting a consistent data layout is essential for the user to understand and use an application interface.

Developing Form Style Windows

Authors: Mark Bradac, Becky Fletcher

Lucent Technologies, Naperville, Illinois

mark.g.bradac@lucent.com, rebecca.l.fletcher@lucent.com

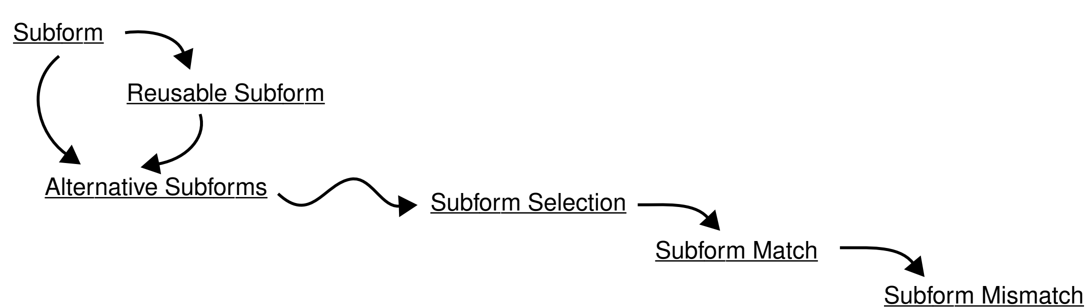
1.0 INTRODUCTION

A Graphical User Interface (GUI) consists of a window, or collection of windows, that contain interface widgets. The function of a widget is to enable the application user to view, create, update, delete, or invoke operations on application data. (Example widgets include: input fields, lists, action buttons, etc.) Often, a particular view of application data is referred to as a form. A form serves to organize a collection of widgets to perform some sort of operation on data, e.g., to collect, view or edit. Typically, there is one window (or the appearance of a window) per form.

Non-trivial applications may employ dozens, even hundreds, of specialized forms to present different views of the same data to different users of the application, e.g., one user may need to be able to create, update, and delete a customer billing record while another may be permitted only to view a subset of the captured information. As a consequence, the differences between various forms are often minor, e.g., the widgets for viewing customer sensitive information may be eliminated on all but a few user privileged forms.

The Window Display pattern language addresses how subcanvases can be employed to manage application form construction and reuse while at the same time minimizing the number of discrete windows that are needed to be built and maintained. “Subcanvases” are a convention of the VisualWorks ApplicationModel hierarchy.

The language consists of six patterns and is depicted in the following directed acyclic graph:



The Subform pattern is used to decompose a form into smaller sections. Reusable Subform facilities the reuse of a subform on multiple forms. Alternative Subforms allows for the dynamic selection of subforms at run time. The Subform Selection pattern allows a form to select the appropriate subform and use it until it is longer valid. Subform Match is used to facilitate subform selection and Subform Mismatch is used to notify the form when it is no longer valid.