8.16.20.26.34.76.88.94.124.148.176. >> 54.
8.24
16.28.64.
15.07.2015
ВО 22 30
Ш.О.
Е.Б.
М.Ш.
-540
206.218.230. >>108.
10. 30. 84. 124.140.146.188.
2.8.10-14. >>
24-48.
16.
4.6.22.34.40.76.94.118.122 >>0
4.6.12 ->
8.44.68.
24-48.
-24
-8
0
+540
+24
+612
+16
+48
-16
-604
-48

How to design own controls

- Inherit from existing control
- UserControl (compose several in one)
- Programming Microsoft Windows Forms (by Charles Petzold)

DEMO

Demo

1) Petzold\4 Custom Controls\BeepButtonDemo
2) RoundButton

GUI Architectures

- Forms and Controls
  - State
  - Events
  - Data Binding
  - Flow Synchronization

Рисовать схемму на доске

Отображение данных record state / session state / screen state

Mainstream approach to UI design.

The Forms and Controls model provides a design that is easy to understand and makes a good separation between reusable widgets and application specific code

What it lacks, and MVC has so strongly, is Separated Presentation and indeed the context of programming using a Domain Model

GUI Architectures

- Model View Controller (MVC)
  (parts of classic MVC don't really make sense for rich clients these days)
  - Separated Presentation idea
  - Observer Synchronization
  - Presentation Model

Domain element is referred to as the model; The presentation part of MVC is made of the two remaining elements: view and controller;

-The controller's job is to take the user's input and figure out what to do with it. So the first part of reacting to the user's input is the various controllers collaborating to see who got edited. In the case that's it's the [SOME] text field so that text field controller would now handle what happens next.

-all the views and controllers observe the model

-controller is very ignorant of what other widgets need to change when the user manipulates a particular widget

-Observer behavior is hard to understand and debug

Summary:

Make a strong separation between presentation (view & controller) and domain (model) - Separated Presentation.

Divide GUI widgets into a controller (for reacting to user stimulus) and view (for displaying the state of the model). Controller and view should (mostly) not communicate directly but through the model.

Have views (and controllers) observe the model to allow multiple widgets to update without needed to communicate directly - Observer Synchronization. Source: http://www.martinfowler.com/eaaDev/uiArchs.html

GUI Architectures

- Model-View-Presenter (MVP)
  - No view/controller separation
  - Observer Synchronization
  - Supervising Controller / Passive View

The **view** of MVP is a structure of these widgets. It doesn't contain any behavior that describes how the widgets react to user interaction.

The active reaction to user acts lives in a separate **presenter** object. The fundamental handlers for user gestures still exist in the widgets, but these handlers merely pass control to the presenter.

The presenter then decides how to react to the event. Does actions on the model, which it does by a system of commands (Command pattern)

As the **Presenter** updates the **model**, the **view** is updated through the same Observer Synchronization approach that MVC uses.

One of the variations in thinking about MVP is the degree to which the presenter controls the widgets in the view

Supervising Controller, where the view handles a good deal of the view logic that can be describe declaratively and the presenter then comes in to handle more complex cases

Summary

User gestures are handed off by the widgets to a Supervising Controller.

The presenter coordinates changes in a domain model.

Different variants of MVP handle view updates differently. These vary from using Observer Synchronization to having the presenter doing all the updates with a lot of ground in-between.