PROGRAMACAO DE ELITE

REQUISITO DADO É CÓDIGO IMPLEMENTADO

1- Introdução

Olá galera !
Esse tutorial é um resumão da minha palestra “Programação de Elite – Requisito dado é código implementado”.

Vou passar os tópicos mais importante para você se achar nos exemplos e seguir o fluxo de raciocínio que criei para mostrar as técnicas envolvidas.

Nosso texto se divide em 3 partes:
- Controle de versão;
- Desenvolvimento (Técnicas e padrões)
- Testes Unitários (Introdução ao DUnitX)

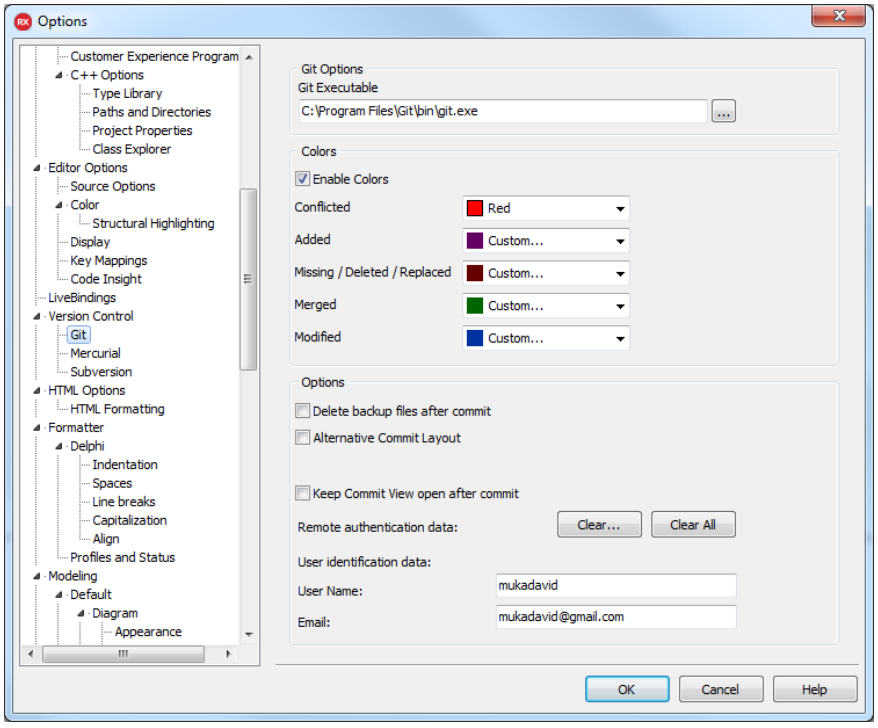
2- Controle de versão

O Delphi Seattle apresenta integração com 3 modelos de versionadores.
- Subversion
- Git
- Mercurial

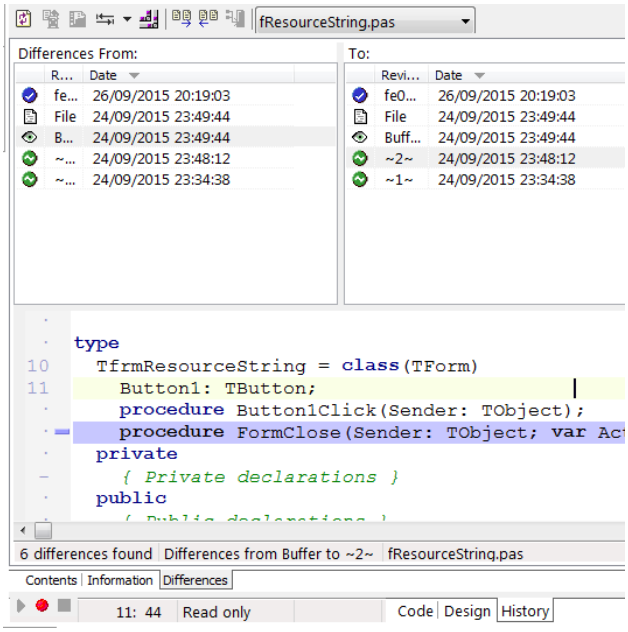
Os “clients”, precisão ser instalados na máquina, não é feita nenhuma instalação deles junto a instalação do Delphi .

O Git e o Mercurial precisam ser configurados manualmente.
Já o Subversion a integração é automática.

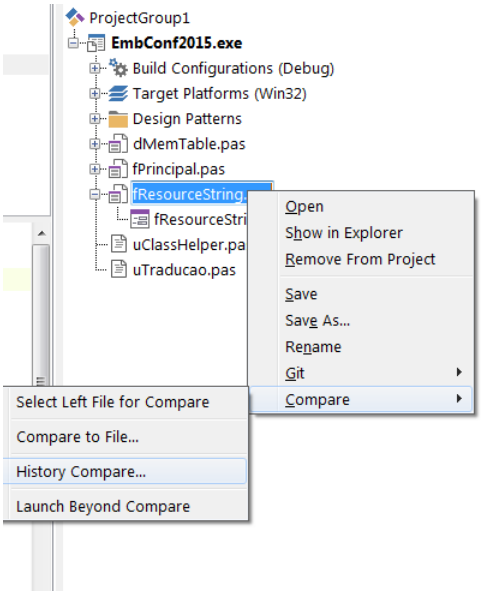
Para configura-los vamos
Tools > Options > Version Control >



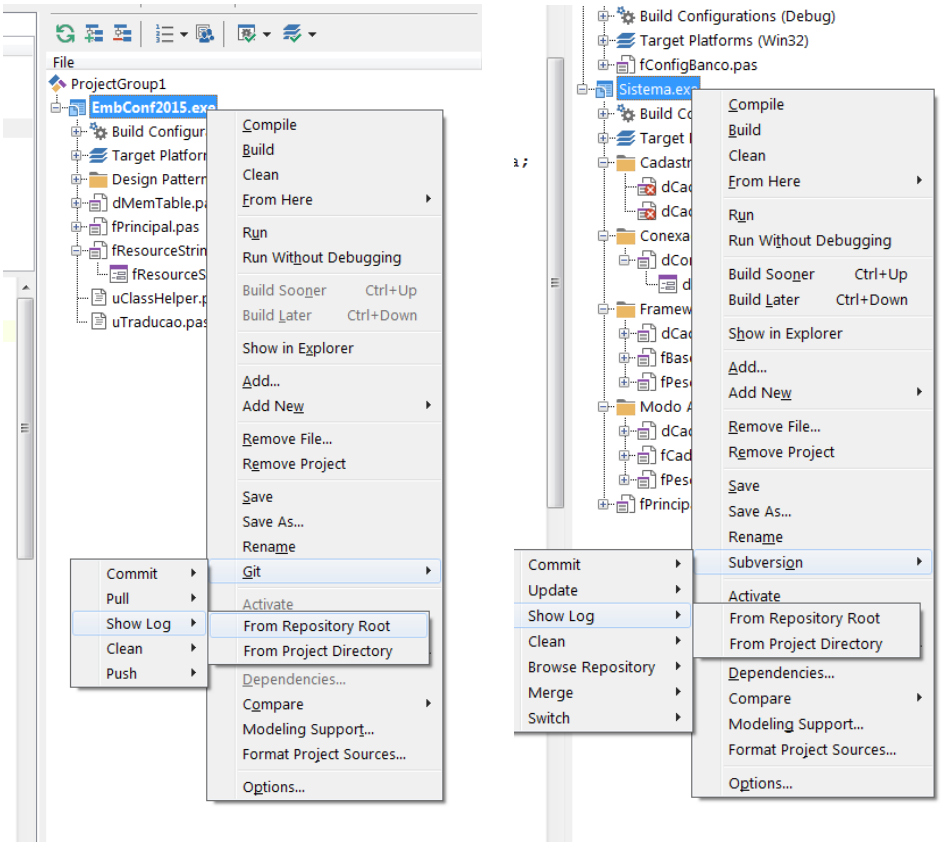
Além dos versionadores temos os históricos e comparações de fontes.
Uma das opções de comparação de arquivo continua sendo a a janela padrão de histórico e comparação do Delphi.



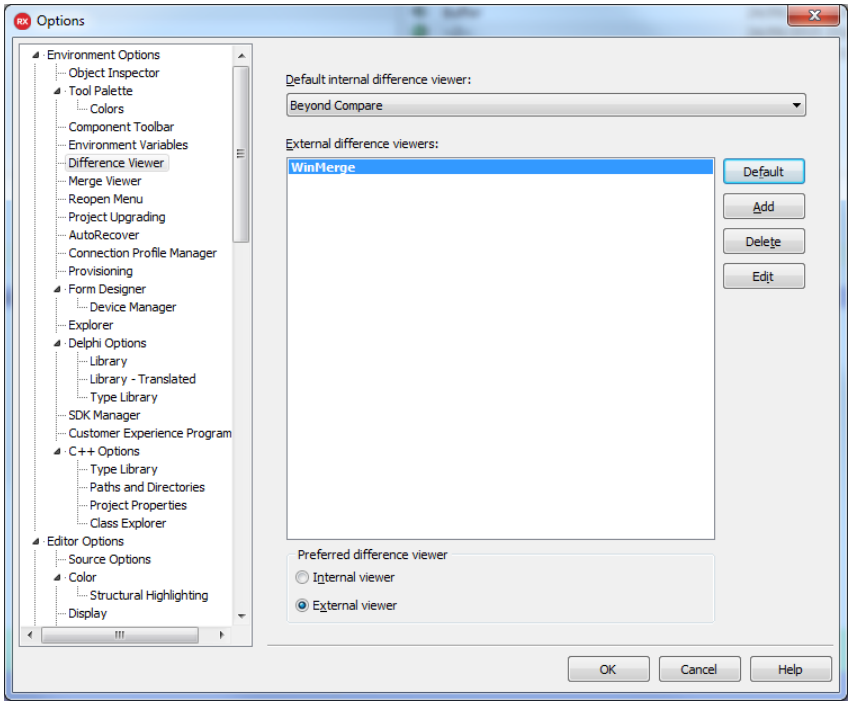
O Delphi também vem equipado com O Beyond Compare Lite 4.
Uma Ferramenta de comparação e merge de arquivos muito conhecida na comunidade.



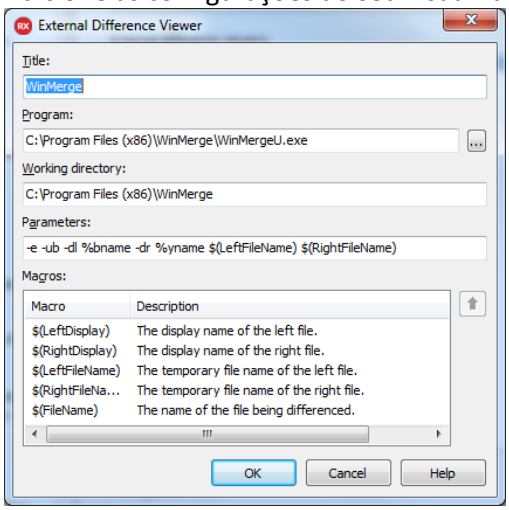
Toda vez que você associar seu projeto a um versionador você poderá fazer commit, update e merge de seus fontes diretamente da IDE.



Por padrão as comparações de histórico serão feitas pelo Beyond Compare mas é possível alterar. Por exemplo, eu gosto muito de usar o WinMerge, além de ser muito intuitivo e free, ele se integra totalmente ao tortoise, client de versionador.
Para fazer as configurar um comparador externo va em Environment Options > Difference Viewer:



Adicione as configurações do seu visualizador



Para controle de versão gratuito temos dois servidores que disponibilizam este serviço:
- bitbucket.org , que suporta tanto Git quanto Mercurial
- assembla.com , que suporta Git e Svn.

3- Orientação Objetos

"Quando um programador entra em um código legado para refatorar,
Não adianta achar que vai dar para apagar tudo e refazer do Zero.
Existe muito código inocente, que foi escrito corretamente e não merece ser apagado."

"O Programador de Elite não entra em código fonte apagando tudo, ele entra com estratégia, progride de bloco em bloco, método por método."

E para conseguir fazer isso é preciso conhecimento e muito treino.

Quando eu falo de Orientação Objetos pode ter certeza que não maior parte das vezes o pessoal não passa nem pelo o básico.

Vamos lá!

Os 4 paradigmas da Orientação Objeto:

Abstração: A capacidade de simular o mundo real. Um objeto deve possuir identidade (Instância), características (Propriedades) e podendo realizar ações (métodos);

Herança: Capacidade de um objeto receber as características de outro e estende-las;

Encapsulamento: Poder encaixotar tudo, não há necessidade de conhecer certos níveis da hierarquia das classes apenas conhecer suas chamadas;

Polimorfismo: Permite que objetos possam ser representados por suas classes bases, mas mantendo a funcionalidade original.

Vamos pegar um pequeno exemplo problema,

Eu tenho uma classe que representa uma ficha de usuário e eu preciso de persisti-la em um arquivo.

Exemplo1.pas

Precisamos que seja possível exportar também para Xml, Json
O que você faz?
Cria um tipo para que seja selecionado o tipo por parâmetro e usa como condição?

Exemplo2.pas

Como a gente faz?

Tem que progredir com calma...
Fatiou! Passou!
Refatora...

Exemplo3.pas

Ok!
Continuamos a adicionar cada vez mais métodos na nossa classe?

Um dos cuidados que devemos ter é que nossas implementações devam ser simples, e pequenas.
Alguém poderia sugerir que possamos especializar nossa classe e utilizar polimorfismo.

Exemplo4.pas

O Exemplo é interessante, mostra bem como é o funcionamento polimórfico de uma classe, mas eu tenho uma dependência muito grande de responsabilidades.

E se eu quiser criar um objeto TFichaAdministrador que será herdado de TFichaUsuario, como eu faço para exportá-lo?

4- Padrões de projeto

“Fazer uma programa funcionar e ter um código bem feito são coisa bem diferente.”

Ultimamente existe uma crescente quantidade de programadores preocupados em utilizar padrões de projetos e outros princípios de programação.

Muito tem se falado sobre o 5 princípios SOLID.

“Os princípios SOLID são cinco princípios básicos de programação e design orientados a objetos, introduzidos por Uncle Bob no início de 2000.
Que se aplicados em conjunto, podem diferenciar um desenvolvedor, tornando-o capaz de escrever um código extensível, coeso e de fácil manutenção. ” (Peguei essa definição na internet, excelente!)

E para os gestores? Fácil manutenção, quer dizer sistemas mais baratos, entregas rápidas e menos erros!

Um desses princípios é o do princípio da responsabilidade única.
Esse princípio afirma que uma classe deve ter apenas um único motivo para **mudar**.

Quando vemos nossos exemplos anteriores, observamos que toda vez que vamos alterar alguma coisa referente à exportação estamos alterando nossa classe de ficha.
E nessa classe seu único motivo para mudar seria uma alteração na sua estrutura de dados.

Sendo assim vamos criar uma nova classe que vai ser responsável por efetuar a exportação dos dados de TFichaUsuario.

Exemplo5.pas

No Exemplo5.pas criamos então a classe TFichaUsuario, e implementamos suas respectivas herança em Exemplo5_1.pas.
O Polimorfismo continua funcionando, mas em uma classe separada que não modifica a TFichaUsuario toda vez que precisarmos alterar a exportação.

Desta forma fazemos o desacoplamento de uma responsabilidade,
A responsabilidade de exportar.
Assim diminuímos o acoplamento e aumentamos a coesão, e os objetos passam a ter uma única responsabilidade.

Outra coisa que podemos fazer aqui é aplicar um outro pattern conhecido como factory.
Ao invés de ter que montar um case no momento da chamada da exportação, nós criamos uma classe que vai ser responsável por isso.

Exemplo6.pas

Na unit Exemplo6_1 temos a classe TExportaFichaUsuarioFactory, que será encarregada de nos retornar a instância do objeto conforme o tipos informado.

Será que da para melhorar?

Outro princípio SOLID é o principio da inversão da dependência (DIP).

Alguns conceitos sobre DIP que achei por ai:

"Código contra abstrações não implementadas".

"Sempre dependa de uma interface, não uma implementação."

"Módulos de alto nível não devem depender de módulos de baixo nível, ambos devem depender de abstrações. Abstrações não devem depender de detalhes. Os detalhes devem depender abstrações".

As implementações de classes seguem certo fluxo, partem de uma abstração para algo concreto. Imaginem que na Vcl tudo parte de TObject, e depois vai seguindo... passa por TPersistent, TComponet até virar um TEdit, no caso, uma classe mais concreta.

Quando criamos uma classe que faz um, digamos, um serviço para outra, muitas vezes por conveniência, fazemos o uso da implementação mais conqueta de uma hierarquia de classes, isso torna as possibilidades de uso dessa classe de serviço limitada a uma única classe, em nosso exemplo, TFichaUsuario.

A ideia é inverter a dependência dentro deste fluxo, você não deve depender de classes concretas e sim de níveis mais abstratos, e neste caso quem melhor trabalham são as interfaces, não há nada mais abstrato que isso!

Sendo assim vamos originar nossas classes a partir de interfaces... e faremos uma injeção de dependência pela classe TGerenciadorFicha. Que ficará responsável por ligar o interface IFichaUsuario e IExportaFichaUsuario.

Exemplo7.pas

Assim a classe TGerenciadorFicha fica encarregada das execuções da regra de negócio, mas a regra pode ser implementada por qualquer classe que implemente as interfaces.

Como eu não posso criar campos em Interfaces eu declaro os métodos de leitura, os gets!

Se você quiser que sua classe tenha suporte a FichaUsuario basta implementar a interface em algum nível da hierarquia.

E para finalizar podemos simplificar a chamada do método exportar do TGerenciadorFicha, através de um recurso da linguagem chamado class helper.

Implementamos um helper para encapsular a criação, o método exportar e a destruição do objeto gerenciador. Isso sem fazer dependência na estrutura de interfaces

Poderia resolver com um facade, uma vez que a classe é minha não preciso usar um helper, a ideia dele é justamente permitir que eu adicione funcionalidades em classes que não posso modificar. Mas serve como um ótimo exercício!

Pensando dessa forma pode até parecer complicado, fizemos uma série de alterações, falamos de vários princípios de desenvolvimento...
Mas tudo isso é uma questão de treino...

Vocês devem treinar, exercitar os Patterns, peguem uma semana para cada pattern...
Crie classes, jogue com as implementações.

Você deve ser capaz de fazer esse tipo de implementação sem muita refatoração.
É como dirigir! Você acelera, pisa na embreagem, muda de marcha, solta a embreagem, aceleram, reduz, liga o pisca...

E isso só é possível treinando.
Músicos, praticantes de artes marciais ou qualquer outro exporte treinam para otimizar suas habilidades. A Ferrari não troca pneus em segundos por mero acaso.
“A prática leva a perfeição”

5- Testes

EmbConf2015DUnitXTest.dproj

Ninguém gosta de testar!

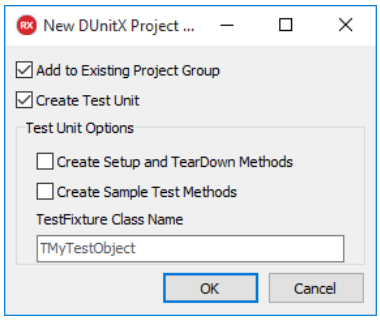
Mas escrever testes também é um excelente exercício de programação, e muito necessário!

Vou apresentar o DUnitX!

Um framework de teste que vem com incorporado ao Delphi.

Para construirmos um projeto de teste!

Va em File -> New -> Other -> Delphi Projects -> DUnitX -> DUnitX Project



Informamos no wizard o nome da classe de teste que queremos criar e se quiser uma colinha marque a opção Create Sample.

Para fazermos nossos testes unitários basta implementarmos a classe de teste, marcando os métodos com o atributo [Test] para executar as chamadas ou [TestCase], neste caso para que eu faça a passagem de parâmetros de teste.

Além disso também posso criar meus próprios atributos de teste, basta herdar de TestCaseAttribute.
Mas tem um macete, os valores recebidos pelos parâmetros do atributo devem ser gravados na variável FCaseInfo em seu campo values, e devem ser alimentados na ordem em que se da a passagem de parâmetros para o método de teste.

uTesteExemplo7.pas

O DUnitX não possui uma IDE visual, ele não é gráfico.
Isso se deve pelo fato dele ser um framework com muitas possibilidades de expansão.

Identifiquei a interface ITestLogger, que recebe os logs de teste, e implementei uma classe a partir desta interface para reportar o log para um formulário.
EmbConf2015DUnitXMonitorTest.dproj

Instancio o framework de teste no Button1 do formulário, e ainda dou uma brincada com métodos anônimos para capturar os eventos do log.

FMonitorDeTeste.pas

6- Conclusão
Encerramos por aqui!
Duvidas, críticas, apontamentos inteligentes são sempre bem vindos