1. Переменные, наследуемые другими классами от родителя должны быть protected, а не public, имена переменных должны соответствовать стандартам.

Было:

public NewSystemContext newSystemContext;

public NewDatabase newDatabase;

Стало:

protected NewSystemContext \_newSystemContext;

protected NewDatabase \_newDatabase;

1. Вынести все статические переменные в константы. Однако такой момент был всего 1 и необходимость в нем пропала после обновления DBF исходников.

Следующий кусок кода был удален из проекта:

int crace\_id = 0;

int.TryParse(item.Crace, out crace\_id);

if (crace\_id <= 18179)

1. Использование Nullable

Было:

\_newRaceResult.HorseId = NewSystemContext.Horses.FromCache().FirstOrDefault(x => x.OldSystemId == item.Chorse).Id;

Стало:

\_newRaceResult.HorseId = NewSystemContext.Horses.FromCache().FirstOrDefault(x => x.OldSystemId == item.Chorse)?.Id;

Суть в том, что HorseId и подобные поля могут принимать null, а если убрать nullable из запроса, может возникнуть исключение. В данном же случае, вместо исключения будет возвращено null и поле будет успешно записано.

1. Вместо длинных типов переменных использовать var (если тип заранее известен, что и есть в большинстве случаев):

Было:

List<NRacehits> result = NewDatabase.Racehits.ToList();

List<RaceResult> newData = new List<RaceResult>();

Стало:

var result = NewDatabase.Racehits.ToList();

var newData = new List<RaceResult>();

1. Для инициализации БД теперь используется using в родительском классе:

public async Task Start()

{

using (NewDatabase = new NewDatabase())

using (NewSystemContext = new NewSystemContext())

{

await WorkAsync();

}

}

Тем самым мы избавили дочерние классы от необходимости создавать собственный экземлпяр контекста и разрушать его (убрав много избыточного кода). Для каждого вызова WorkAsync будет создаваться свой собственный контекст и разрушаться по мере завершения работы этого контекста.

1. Изменены некоторые имена методов и классов. Например, было UpdateProgress, стало NewDataSynchronizator., ProgramManager -> DataImporter, DataImporter.View() -> DrogramManager.DoWork()
2. Классы, связанные с БД названы более понятно:

NewDatabase -> DbfDatabase,

NewSystemContext -> NewSqlDatabaseContext

1. Добавлен и настроен Nlog
2. Там где это возможно, лямбда из Where() перенесена SingleOrDefault/FirstOrDefault

Стало:

DbfDatabaseContext.Slhirid.FirstOrDefault(x => x.Crider == item.OldSystemId);

1. Код SlhorsesImporter.cs оптимизирован, количество запросов уменьшено.

Например отец и мать теперь получаются одним запросом:

var newParentInfo = NewSqlDatabase.Horses.FromCache()

.Select(x => new {x.Id, x.Name, x.OldSystemId})

.Where(x => x.OldSystemId == oldInfo.Cmoth || x.OldSystemId == oldInfo.Cfath)

.ToList();

Теперь сохранение данных айтема не происходит по принципу «поле – запрос». Сначала каждому полю присваевается новое значение и только после изменения всех полей происходит апдейт.

1. Добавлен AsNoTracking() для мелких словарей, а также для запросов, данные которого не изменяются с течением времени а используются только лишь для получения информации.

Например:

var horseList = await ResultData.Horses.AsNoTracking().Select(x => new { x.Id, x.OldSystemId}).ToListAsync();

\_riders = ResultData.Riders.AsNoTracking().FromCache().ToList();

\_penalties = ResultData.LookupRacePenalties.AsNoTracking().FromCache().ToList();

1. Импорт из DBF2SQL Должен быть параллелен лучше, чем на данный момент. До рефакторинга он был асинхронным – каждый из методов выполнялся последовательно, хоть и Task.WhenAll давал прирост в производительности, этот способ импорта не был ожидаемым.

Было:

return Task.WhenAll(

\_horhippoImporter.StartImportAsync(),

\_measureImporter.StartImportAsync(),

\_slcolorImporter.StartImportAsync(),

\_slhorsesImporter.StartImportAsync(),

\_slstockImporter.StartImportAsync(),

\_addnameprizeImporter.StartImportAsync(),

\_applicatImporter.StartImportAsync(),

\_moneycoefImporter.StartImportAsync(),

\_podachaImporter.StartImportAsync(),

\_prizedistrImporter.StartImportAsync(),

\_programsImporter.StartImportAsync(),

\_racehitsImporter.StartImportAsync(),

\_racesImporter.StartImportAsync(),

\_slcategImporter.StartImportAsync(),

\_slcourseImporter.StartImportAsync(),

\_slhiprizImporter.StartImportAsync(),

\_slhiridImporter.StartImportAsync(),

\_slonstartImporter.StartImportAsync(),

\_slpenaltImporter.StartImportAsync(),

\_slpositImporter.StartImportAsync(),

\_sltudsImporter.StartImportAsync(),

\_sltrackImporter.StartImportAsync(),

\_stablesImporter.StartImportAsync(),

\_standardImporter.StartImportAsync(),

\_zaezdistrImporter.StartImportAsync(),

\_slhiracImporter.StartImportAsync()

);

Стало:

Parallel.ForEach(\_listServices, x => x.StartImportAsync());

+ В конструкторе объявлено:

\_listServices.Add(new AddnameprizeImporter());

\_listServices.Add(new ApplicatImporter());

\_listServices.Add(new HorhippoImporter());

\_listServices.Add(new MeasureImporter());

…

Для каждого из импортеров. Данный подход позволяет более гибко и быстро внедрить новый импортер в код, добавив всего одну строчку (добавление инициализатора класса в список List). Все остальные действия не требуют вмешательства.

1. Разработан класс LookupManager, автоматически заполняющий списки словарей, которые не будут изменяться с течением времени или представляют собой данные, которых нет в старой базе.

