Geekbrains

**Сайт-руководство по созданию плагинов для программы для 3D моделирования строительных конструкций Tekla Structures с использованием Tekla Open API на языке программирования C#**

Инженер-программист

Маркова О.С.

Москва

2024 г.

Содержание:

Введение

Глава 1. Теоретическая часть

1.2 Tekla Structures

1.3 Application programming interface (API)

1.4 Tekla Open API

1.5 Язык C#

1.6 Некоторые термины языка C#

Глава 2. Что такое плагин?

2.1 Плагин

2.2 Почему был выбран язык C#?

2.3 Три способа разработки графического интерфейса плагина

2.4 Какие бывают плагины в Tekla?

2.5 Обязательные части плагина

Глава 3. Написание плагина

3.1 Пространства имен

3.2 Атрибуты

3.3 Класс Input\_Beam

3.3.1 Метод DefineInput().

3.3.2 Метод Run

3.4 Класс UI\_Data

3.5 Класс Preset

3.5.1 Main\_Panel

3.5.2 Ui\_Class\_CheckBox

3.5.3 TextBox.

3.5.4 OkApplyModifyGetOnOffCancel

3.5.5 SaveLoad

3.6 Метод GetValuesFromDialog()

3.7 Метод SetValuesFromData

3.8 Метод Main

Глава 4. Сайт.

Заключение

Список литературы и ресурсов

**Введение.**

В современном мире строительного проектирования требования к скорости и качеству работы постоянно растут. Программное обеспечение для 3D моделирования, такое как Tekla Structures, играет ключевую роль в удовлетворении этих требований, предоставляя инженерам мощные инструменты для создания детализированных конструкционных моделей. Однако, несмотря на широкий спектр функциональных возможностей, стандартный набор инструментов иногда не может удовлетворить специфические потребности проекта. В таких случаях на помощь приходит Tekla Open API, который позволяет расширять и адаптировать функциональность Tekla Structures с помощью пользовательских плагинов.

Целью данной дипломной работы является разработка руководства по созданию плагинов для Tekla Structures на языке программирования C#. Руководство будет служить практическим пособием для инженеров и разработчиков, желающих научиться создавать индивидуальные решения для улучшения рабочих процессов в программе и в дальнейшем будет размещено в сети интернет для общего пользования.

Актуальность работы подтверждается не только текущими тенденциями в строительной отрасли, но и потребностью рынка в интструментах для кастомизации и автоматизации рабочих процессах.

При написании данной работы консультантом по Tekla Structures выступил Евгений Владимирович Бетеня - BIM-координатор АО "Атомэнергопроект".

**Теоретическая часть.**

**Tekla Structures.**

Tekla Structures **-** это программное обеспечение для информационного моделирования зданий и сооружений промышленного и гражданского строительства. Оно способно моделировать конструкции из различных строительных материалов, включая сталь, бетон, дерево и стекло.

Tekla Structures позволяет проектировщикам и инженерам выполныть следующие задачи:

- моделировать конструкцию здания и его компоненты с помощью трёхмерного моделирования;

- создавать двухмерные чертежи;

- получать доступ к информации о здании или сооружении.

Программное обеспечение (ПО) позволяет пользователям создавать 3D-модели конструкций из бетона или стали и управлять ими, а также направляет их на протяжении всего процесса от концепции до изготовления, имеет автоматизированный процесс создания рабочих чертежей. Множество процессов детализации выполняется в ПО автоматически. ПО доступно в различных конфигурациях и локализованных средах.

В работе мы используем Tekla версии 2021 года, так как именно она является последней официально доступной лицензионной версией для большенства компаний на территории России. Стоит отметить, что большинство пользователей Tekla используют Tekla 2019, 2020 и 2021 версии.

**Application programming interface(API)**

API - это программный интерфейс приложений, набор инструкций, который позволяет разным приложениям общаться между собой. API может быть:

- У языка программирования, чтобы разные функции правильно взаимодействовали между собой. Каждая функция тут выступает «приложением», а API становится набором инструкций для правильного вызова этих функций.

- У операционной системы, чтобы программы могли извлекать из неё данные и при необходимости изменять настройки операционной системы. При разработке приложения для Windows, Linux или Android нужно знать API этой системы, чтобы работать с файлами и графикой.

- У веб-сервисов, чтобы другие веб-сервисы и программы могли к ним подключаться и работать совместно.

Принцип работы API прост:

- Сторонний разработчик пишет какое-то своё приложение, функцию, операционную систему.

- Он составляет API - набор правил, на основе которых другие разработчики могут использовать его разработку тем или иным способом.

- Вы получаете этот API - он либо лежит в публичном доступе, либо открывается по запросу для клиентов.

- Вы вызываете API внутри своего приложения и пользуетесь нужными вам функциями.

При этом вам не нужно знать, как работает API-функция, которую вы вызываете. Она может быть очень сложной и содержать тысячи строк кода — а вы будете вызывать её всего одним действием и получать нужный результат.

**Tekla Open API**

Tekla Open API - это интерфейс программирования приложений, который позволяет разрабатывать собственные приложения и дополнительные функции для Tekla Structures. Он реализуется с использованием технологии Microsoft .NET. Приложения, разработанные с помощью Tekla Open API, называются расширениямиили плагинами.

С помощью Tekla Open API можно:

- Записывать и запускать действия пользовательского интерфейса, автоматизировать рутинные задачи, например, создание ежедневных отчетов.

- Создавать инструменты автоматизации для объектов.

- Интегрировать Tekla Structures с другим программным обеспечением.

- Создавать новую функциональность.

**Язык С#**

C# - современный объектно-ориентированный и типобезопасный язык программирования. C# позволяет разработчикам создавать разные типы безопасных и надежных приложений, выполняющихся на платформе .NET. C# относится к широко известному семейству языков C.

C# - объектно-ориентированный, ориентированный на компоненты язык программирования. C# предоставляет языковые конструкции для непосредственной поддержки такой концепции работы. Благодаря этому C# подходит для создания и применения программных компонентов. С момента создания язык C# обогатился функциями для поддержки новых рабочих нагрузок и современными рекомендациями по разработке ПО.

Вот лишь несколько функций языка C#, которые позволяют создавать надежные и устойчивые приложения:

- Сборка мусора автоматически освобождает память, занятую недостижимыми неиспользуемыми объектами.

- Типы, допускающие значение null, обеспечивают защиту от переменных, которые не ссылаются на выделенные объекты.

- Обработка исключений предоставляет структурированный и расширяемый подход к обнаружению ошибок и восстановлению после них.

- Лямбда-выражения поддерживают приемы функционального программирования.

- Синтаксис LINQ создает общий шаблон для работы с данными из любого источника.

- Поддержка языков для асинхронных операций предоставляет синтаксис для создания распределенных систем.

- В C# имеется Единая система типов. Все типы C#, включая типы-примитивы, такие как int и double, наследуют от одного корневого типа object. Все типы используют общий набор операций, а значения любого типа можно хранить, передавать и обрабатывать схожим образом. Более того, C# поддерживает как определяемые пользователями ссылочные типы, так и типы значений.

- C# позволяет динамически выделять объекты и хранить упрощенные структуры в стеке.

- C# поддерживает универсальные методы и типы, обеспечивающие повышенную безопасность типов и производительность.

- C# предоставляет итераторы (перечислители), которые позволяют разработчикам классов коллекций определять пользовательские варианты поведения для клиентского кода.

- C# поддерживает Управление версиями, чтобы обеспечить совместимость программ и библиотек с течением времени.

**Термины из языка C#, используемые в данной работе:**

- Атрибут

В C# разрешается вводить в программу информацию декларативного характера в формате атрибута, с помощью которого определяются дополнительные сведения (метаданные), связанные с классом, структурой, методом и т.д. Например, в программе можно указать атрибут, определяющий тип кнопки, которую должен отображать конкретный класс. Атрибуты указываются в квадратных скобках перед тем элементом, к которому они применяются. Следовательно, атрибут не является членом класса, но обозначает дополнительную информацию, присоединяемую к элементу

- Делегат

Представляет собой объект, который может ссылаться на метод. Следовательно, когда создается делегат, то в итоге получается объект, содержащий ссылку на метод. Кроме того метод можно вызвать по этой ссылке. Иными словами, делегат позволяет вызывать метод, на который он ссылается. Один и тот же делегат может быть использован для вызова разных методов во время выполнения программы, для чего достаточно изменить метод, на который ссылается делегат. Таким образом, метод, вызываемый делегатом, определяется во время выполнения, а не в процессе компиляции. В этом и заключается главное преимущество делегата. Это очень удобно для создания базовой конструкции, допускающей подключение отдельных программных компонентов. Также делегаты поддерживают события.

- События

Представляют собой автоматическое уведомление о том, что произошло некоторое действие. События действуют по следующему принципу: объект, проявляющий интерес к событию, регистрирует обработчик этого события. Когда же событие происходит, вызываются все зарегистрированные обработчики этого события. Обработчики событий обычно представлены делегатами.

- Рефлексия

Это средство, позволяющее получать сведения о типе данных. Термин рефлексия, или отражение, происходит от принципа действия этого средства: объект класса Type отражает базовый тип, который он представляет. Для получения информации о типе данных объекту класса Type делаются запросы, а он возвращает(отражает) обратно информацию, связанную с определенным типом. Рефлексия является эффективным механизмом, поскольку она позволяет выявлять и использовать возможности типов данных, известные только во время выполнения. Ведь получив сведения о типе данных, можно сразу же вызвать его конструкторы и методы или воспользоваться его свойствами. Следовательно, рефлексия позволяет использовать код, который не был доступен во время компиляции.

- Инкапсуляция

Идея инкапсуляции состоит в том, что вы скрываете реализацию ваших классов от пользователей (и программистов), предоставляя им для работы интерфейсы ваших классов. Также это подразумевает скрытие данных при помощи идентификаторов доступа (public, private, protected).

**Что такое плагин?**

**Плагин**

При разработке компьютерного софта программисты не вкладывают в программу весь возможный функционал, а, как правило, ограничиваются базой и необходимыми дополнениями в расширенных версиях. Это позволяет сделать приложение достаточно легким в использовании, предназначенным для разных ПК. При этом интерфейс рабочей зоны не перегружен и интуитивно понятен. Ведь как только появляются всевозможные значки, функции, сразу становится сложно в них ориентироваться. Но людям для реализации своих задач требуются другие возможности, для этого сторонние разработчики пишут разные надстройки.

Англоязычный термин plugin пришел в русский язык без изменений, он обозначает подключаемый модуль. Фактически небольшой программный код предназначен для установки на конкретное обеспечение. Дополнение позволяет расширять функционал ПО или исправлять изначальные баги. Задача надстройки - сделать программу лучше, переквалифицировать ее в узкоспециализированную, добавив функции, решающие конкретные проблемы.

Отличие плагина от полноценного софта в том, что он не может использоваться отдельно, как самостоятельный элемент, а обязательно устанавливается сверху, «надстраивается». И работает внутри программного продукта, на основе его интерфейса, базовом коде. База легко может функционировать без каких-либо модулей.

Если объединить всё вышесказанное, плагин — это независимо компилируемый программный модуль, который динамически подключается к основной программе и предназначен для расширения и/или использования её возможностей.

Плагины обычно выполняются в виде библиотек общего пользования. Для возможности подключения плагинов разработчик основного приложения должен предусмотреть в нём некоторый программный интерфейс, а также минимальные возможности по управлению набором плагинов.

Примеры плагинов:

- фильтры для растровых графических редакторов;

- поддержка форматов файлов для звуковых и видеопроигрывателей, пакетов офисных приложений, программ обработки звука и графики;

- плагины для почтовых программ (спам-фильтры, плагины для проверки писем антивирусом);

- плагины для настройки параметров компьютерных игр.

Чаще всего plugin пишут для сферы онлайна – для браузеров, сайтов. Также распространены «моды» или модификации, то есть такие же дополнения, но для компьютерных игр. Оффлайн-программы тоже иногда подвергаются апгрейду, например, графические редакторы или разный софт для трехмерного проектирования.

**Почему был выбран язык C#?**

Язык API Tekla написан с использованием технологии NET. На платформе .NET Framework можно писать на нескольких языках. С# является наиболее распространенным среди этих языков, поэтому для написания плагинов мы будем использовать именно его. Помимо C# можно было использовать Python, но по сравнению с C#, результатом компиляции на Python будут очень большие файлы. Аналогичные файлы на С# и Python будет весить 200кб и 20мб соответственно. Следовательно будет использоваться больше памяти. А если учесть, что плагинов используется много и каждый будет занимать много места, то будет значительно страдать скорость выполнения.

Самая нижняя версия .NET Framework для взаимодействия с API Tekla - 4.6, самая - верхняя 4.8.1. Эта версия является последней, дальнейшего развития платформы .NET Framework не происходит, но его смену пришла .Net Core. API Tekla на .Net Core работать не будет. .NET Framework входит в состав операционной системы Windows, но по-умолчанию она устанавливается довольно низкой версии. Установщик Tekla доустанавливает недостающие для работы модули, в том числе обновляет версию .NET Framework до 4.8.1.

Информация об использовании API Tekla является открытой и на официальном сайте Tekla можно получить первоначальную информацию обо всех классах и методах, которые применяются. Для удобства разработки Visual Studio можно скачать NuGet-пакет, он есть для разных версий Tekla и находится в свободном доступе. 2021 версия Tekla последняя, к которой можно его получить. NuGet-пакет - это архив, в котором хранятся DLL библиотеки (англ. Dynamic Link Library — «библиотека динамической компоновки», «динамически подключаемая библиотека»), его можно получить через интерфейс Visual Studio, а можно скачать отдельно с других ресурсов. Visual Studio его разархивирует в специальную область в проекте, для того, чтобы эти библиотеки можно было использовать и сам добавляет в проект ссылки на библиотеки, которые понадобятся для работы. В Visual Studio можно настроить репозиторий для NuGet-пакетов, он может быть как сетевой, так и локальный.

На сайте Tekla есть раздел "Помощь разработчику", содержащий много примеров, которые можно использовать при написании своих плагинов. Примеры не всегда являются понятными по написанию, так как часть кода генерируется с помощью программы Visual Studio, в результате чего осуществляются бессмысленные действия по связыванию тех объектов, которые не требуют связывания. Остановимся на этом вопросе немного подробнее. Интерфейсы можно разрабатывать с помощью встроенных инструментов. Например, графический интерфейс можно собирать с помощью графического инструментария Visual Studio. На макете приложения можно мышкой размещать элементы управления, менять местоположение элементов, привязывать на эти элементы разработчики событий. Но результатом всего этого будет очень большой код, в котором не будет понятно, что и как работает. Все кнопки нужно привязывать к каким-то данным из плагина. Плагин, как единица программы, имеет свои атрибуты и эти атрибуты отвечают за какие-то части алгоритма. Элементы интерфейса должны влиять на эти атрибуты, должна происходить привязка атрибутов к интерфейсу. Если это делать с помощью визуального конструктора, то привязка остается за кадром, то есть остается неясно, что к чему привязывается. Если же писать код самому, то привязка становится "прозрачной" и становится понятно, что конструктор связывает те элементы, которые связывать бесполезно, то есть программа совершает лишние действия, образуется лишний код. Если самому с помощью кода собирать интерфейс, он получается лаконичнее, компактнее и понятнее.

**Три способа разработки графического интерфейса:**

1. Описать текстовый файл в формате .inp. Этот формат используется Tekla для описания атрибутов. Он содержит определенный синтаксис, который позволяет описать тип данных и формат данных (текстовый или цифровой ввод). Это самый быстрый, но не самый гибкий способ. Он позволяет реализовать определенные особенности интерфейса, но не полностью. Например, скрытие полей, но оно работает не совсем корректно и может выдавать ошибки. Главное неудобство этого способа состоит в том, что проверить работает ли сам плагин и его интерфейс, можно только запустив плагин.

Самые распространенные ошибки при написании плагина с использованием встроенного графического интерфейса Tekla:

**-** При создании атрибута для плагина, нужно помнить, что максимально допустимая длина для имени определяемого пользователем атрибута составляет 19 символов. Из-за ошибки в количестве символов, атрибуты, привязанные к полям, которые слишком длинные, не работают. И ошибка об этом не появляется.

- У каждого пользовательского компонента есть входной файл (.inp), который определяет содержимое диалогового окна этого компонента. Пользователи могут редактировать входные файлы диалоговых окон вручную в текстовом редакторе, а во встроенном редакторе интерфейсов. При редактировании входного файла необходимо соблюдать осторожность; ошибки могут привести к тому, что диалоговое окно исчезнет. В руководстве для программы можно найти информацию, какую вкладку и для чего нужно использовать.

Особое внимание нужно обратить на то, что вкладка Общие (четвертая вкладка) зарезервирована для предустановленных общих свойств. Переименовать вкладку Общие или добавить на нее дополнительные параметры невозможно.

Получатся, что использование встроенного интерфейса накладывает много ограничений. Использование же нижеперечисленных инструментов способствует расширению функционала программы, хотя при этом и увеличивается скорость работы.

2. Создание интерфейса с помощью Windows Presentation Foundation - WPF. Аналог Windows Forms, система для построения клиентских приложений Windows с визуально привлекательными возможностями взаимодействия с пользователем. Это графическая (презентационная) подсистема в составе .NET Framework (начиная с версии 3.0), использующая язык XAML.

Считается, что этот способ быстрее, так как он напрямую использует видео-карту.

3. Создание интерфейса с помощью Windows Form.

Windows Forms - интерфейс программирования приложений (API), отвечающий за графический интерфейс пользователя и являющийся частью Microsoft .NET Framework. Данный интерфейс упрощает доступ к элементам интерфейса Microsoft Windows за счет создания обёртки для существующего Win32 API в управляемом коде. Причём управляемый код - классы, реализующие API для Windows Forms, не зависят от языка разработки. То есть программист одинаково может использовать Windows Forms как при написании ПО на C#, C++, так и на VB.Net, J# и др.

Мы будем использовать этот способ, так как считаем его лаконичнее(по сравнению с WPF) и быстрее с точки зрения кода. Он более понятен, использует классическое программирование, где ты сам создаешь классы и определяешь их поведение.

Важно, что именно в последних двух способах при вызове графического интерфейса Tekla выдает окно с предупреждением об ошибке интерфейса. Например, при неправильной длине имени, при привязывании элемента интерфейса более одного раза и так далее.

Когда мы разрабатываем плагин, не используя встроенный интерфейс, на разработку тратится больше времени, но можно получить гораздо больше плюсов. Интерфейс получается более настраиваемый и понятный. И хотя на разработку интерфейса с нуля будет потрачено много времени, полученный код можно использовать вторично и постепенно наработать обширную базу типовых решений.

**Какие бывают плагины в Tekla?**

В Tekla есть такой элемент автоматизации, как компонент. Это те действия, которые программа делает вместо пользователя. Способ задания этого алгоритма может быть разный: с помощью внутреннего редактора (в таком случае это пользовательский компонент) или с помощью программных средств (тогда это плагин). То есть существуют плагины, которые мы пишем сами, а есть плагины, реализованные программистами Tekla. И те, и другие совершают какие-либо действия, используя базовые типы объектов.

Плагины бывают предназначенные для использования в модели и чертежные. Они имеют схожую структуру, но используют разные элементы библиотеки и имеют разное предназначение. И те, и другие плагины запрашивают у пользователя ввод данных с помощью мышки в пространстве модели или чертежа, имеют графический интерфейс (поле для ввода), но они создают разные классы объектов. Чертежные плагины создают объекты чертежа в текущем чертеже. Они служат для того, чтобы создавать аннотативные объекты (объекты, которые согласованно масштабируются при изменении масштаба вида чертежа), чтобы дополнять чертеж какими-либо элементами, которые другим способом невозможно дополнить либо это сделать крайне трудоемко. Плагин же позволяет это сделать гораздо быстрее. Также при помощи чертежных плагинов можно создавать графические таблицы. Текла имеет хороший встроенный инструментарий для создания таблиц, но его не всегда хватает и возникают ситуации, когда его нужно доработать или сделать по-другому. Также важно отметить, что не каждый элемент чертежа имеет соответствие с объектом модели. Экземпляр плагина чертежа - это объект чертежа, а экземпляр объекта плагина модели - это экземпляр объекта модели.

Модельные плагины создают объекты для модели (отдельные объекты, сборки, более сложные структуры, объекты связи между другими объектами), основываясь не тех данных, которые вводит пользователь. Кроме того, плагины позволяют взаимодействовать с каталогами из Tekla.Structures.Catalog, читая, редактируя и/или создавая новые значения в каталогах профилей, материалов, арматурных стержней, метизов, комплектов болтов и геометрических форм. Но главная цель модельного плагина - создание и/или редактирование объектов модели (ModelObject).

Цель нашей работы - плагин для модели.

В пространстве имен Tekla.Structures.Model содержатся следующие типы объектов модели (их можно замоделировать с помощью модельных плагинов):

- Assembly (сборка) - класс определяет строительную марку конструкции: объекты, которые соединяются болтами или свариваются вместе в мастерской. Сборка состоит из основной детали и присоединенных к ней вспомогательных сборочных элементов. Количество вспомогательных элементов ограничено 2048. К иерархическим сборкам также могут быть присоединены подсборки, и у них может быть экземпляр родительской сборки.

- BasePoint - класс предоставляет функциональные возможности, связанные с базовыми точками (это различное взаимодействие с разными системами координат).

- Beam - класс балки. Балка имеет одну начальную и одну конечную точки. Самый распространенный элемент Tekla.

- BentPlate - класс гнутой пластины.

- BoltArray - класс определяет группу болтов в форме массива.

- BoltCircle - класс определяет группу болтов, имеющих форму окружности.

- BoltXYList - определяет группу болтов в форме списка координат XY.

- BooleanPart - представляет вырезанную деталь или добавленную.

- Brep(форма) - класс для трехмерных неизменяемых полигональных тел.

- Connection - класс представляет соединение. Соединение - это нечто, соединяющее две или более частей вместе.

- ContourPlate - класс представляет контурную пластину.

- ControlArc - отрисовка дуги в модели.

- ControlCircle - отрисовка окружности.

- ControlLine - отрисовка отрезка.

- ControlPlane - отрисовка плоскости.

- ControlPoint - отрисовка точки.

- ControlPolycurve - отрисовка поликривой.

- CurvedRebarGroup - класс представляет группу арматурных стержней, имеющих изогнутую форму.

- CustomPart - представляет пользовательский компонент типа деталь.

- CutPlane - обрезка плоскостью

- Detail - пользовательский компонент типа узел

- EdgeChamfer - класс определяет способ снятия фаски с кромки детали.

- Fitting - подгонка балок, помогает укорачивать и/или удлинять балки

- Grid - координатная сетка. Обычно алгоритму не требуется с ней взаимодействовать

- LoadArea - задание нарузки по площади

- LoadGroup - группа загружений (относится к расчетам)

- LoadLine - нагрузка, сосредоточенная по линиям

- LoadPoint - нагрузка, сосредоточенная в точке

- LoadTemperature - нагрузка температуры

- LoadUniform - нагрузка, распределяющаяся по произвольному полигону

- LoftedPlate - лофтинговая пластина(полученная в результате сопряжения двух дуг)

- LogicalWeld - представляет группу сварных швов в модели

- PointCloud - облако точек

- PolyBeam - класс полибалки

- PourBreak - шов бетонирования

- PourObject - отлитый элемент

- PourUnit - представляет собой единицу заливки железобетона в модели.

- RadialGrid - радиально-полярная сетка

- RebarGroup - представляет группу арматурных стержней

- RebarGuideline - модификатор арматурного набора - направление армирование

- RebarMesh - арматурная сетка

- RebarSet - арматурный набор

- RebarSplice - представляет соединение двух арматурных элементов.

- RebarSplitter - представляет собой модификатор арматурного набора - линия разделитель

- RebarStrand - представляет предварительно напряженные пряди для бетонных деталей.

- Seam - пользовательский компонент типа стык

- SingleRebar - одиночный арматурный стержень

- SpiralBeam - спиральная балка

- SurfaceObject - обработка поверхности

- Weld - представляет обычный сварной шов в модели.

Пользовательские компоненты Tekla бывают нескольких типов:

- компоненты соединений (Connection), то есть связи между элементами

- компоненты узла (когда есть только один объект ввода) - Detail

- стыки (разновидность соединений) - Seam

- детали (когда нет никаких входных объектов, а есть только входные точки) - CustomPart. Используются, когда нужно поставить в конкретные координаты конкретную сущность.

**Обязательные части плагина:**

1. Плагин должен иметь интерфейс для связи с пользователем. Какой интерфейс будет использоваться, уточняется при создании класса экземпляра плагина в его атрибутах. Далее Tekla либо сама создает ему интерфейс, либо используется функционал операционной системы.

2. Необходимо определить класс, который отвечает за пользовательские переменные плагина, который будет использоваться алгоритмом и будет связан(Bind) с пользовательским интерфейсом.

3. Нужно сделать переопределение двух методов. В базовом классе методы определены, как абстрактные.

Первый метод - метод ввода - DefineInput(), который возвращает тип объекта List<InputDefinition>(может состоять из точки, id объекта либо из списков точек и id). Метод определяет ввод пользователя: то, как плагин требует от пользователя ввести точки, плоскости или объекты. Обычно в нижней левой части программа пишет указания на необходимые от пользователя действия. Чаще всего требуется что-либо указать мышкой.

Второй метод - метод Run. В этом методе и определяется, что делает плагин. У метода Run в аргумантах находится то, что мы определили из метода DefineInput().

Также можно определить метод Main. Cамой Tekla этот метод не нужен. При запуске программы Tekla берет только два метода - метод ввода данных и метод run. Метод Main нужен для того, чтобы протестировать работу плагина, не встраивая его в текущую работу Tekla. Если мы разрабатываем на встроенном интерфейсе Tekla, то сможем протестировать только внутренний алгоритм, то есть метод run. Используя WPF или WinForms, можно сделать так, что у нас сначала запускается графический интерфейс и мы можем его проверить на относительную функциональность, а уже дальше запускается метод ввода. То есть мы можем протестировать плагин, не перезапуская Tekla, что позволяет экономить очень много времени.

Дополнительно в плагине можно определять методы, которые позволяют читать и записывать в сам плагин те значения атрибутов, которые пришли из интерфейса. Плагин, как элемент Tekla, хранит о себе какую-то информацию. Возникают такие ситуации, когда атрибут должен хранить неопределенное значение. На самой платформе Net FrameWork для этого используются специальные типы данных, например, string, которые могут иметь значение null, числовые типы данных, которые могут иметь значение none. Плагин же не умеет хранить значения в таком виде, поэтому он использует крайние значения этих атрибутов(например, самое минимальное или максимальное). Поэтому для алгоритма нужно расшифровывать эти значения и объяснять алгоритму, что это не бесконечные значения, а только неопределенные. В Tekla есть встроенный метод, которые позволяет это сделать - IsDefualt. Дополнительно можно определить метод, который записывает все значения, которые есть, в плагин. Примеры, выложенные в интернете на сайте с документацией, предполагают, что вы должны заново перечислить все атрибуты, которые есть у плагина и проверить их на дефолтные(неопределенные) значения и привести их к типам понятным для платформы NET Framework. Особо тяжелые плагины могут иметь десятки атрибутов. То есть сначала их нужно объявить, привязать к интерфейсу, после чего все тоже самое нужно сделать в методе записи и методе чтения. Чтобы не тратить на это время, может быть разработан более эффективный метод с использованием рефлексии классов. Это позволяет сразу сделать все необходимые действия, независимо от их общего количества. Однако данный метод выходит за рамки нашей дипломной работы из-за повышенной сложности.

Целью нашей работы будет создание простого плагина, который будет запрашивать у пользователя ввод двух точек, и на основании этих двух точек он будет строить балку заранее определенного сечения. Также мы создадим пользовательский интерфейс на основе Windows Forms, через который пользователь может указать, какой используется цвет(класс).

Большую часть плагина можно брать за основу для будущих плагинов, так как большая часть кода является обязательной для создания любого модельного плагина.

Может возникнуть вопрос, для чего вообще нужены плагины в Tekla? Ведь ту же балку пользователь может сделать сам. Вообще все действия, которые делаются с помощью плагинов пользователь может сделать сам. Но Tekla с помощью плагина сделает это значительно быстрее. В этом и суть автоматизации - ускорение действий пользователя. Также плагины решают проблему с внесением изменений в модель. При изменениях в модели плагин может реагировать на эти изменения, так как элементы находятся во взаимосвязи друг с другом и изменение одного объекта, изменяет и другой объект. Tekla не практикует автоналожение связей на макроуровне, то есть сам инженер решает, будет ли в этом месте взаимосвязь между элементами или нет, и если будет, то какая. Плагин же будет работать более точно, но по какому-то определенному алгоритму. Плагины также позволяют более полно раскрыть возможности самой Tekla, очень многое от пользователя спрятано и штатными инструментами некоторые действия произвести невозможно. Например, запросив балку, стандартный отчет выведет информацию о всех пользовательских и системных атрибутах, в то же время, запросив отчет о точке, линии или дуге, стандартный отчет выведет информацию только о положении элемента. Хотя на эти элементы, так же как и на балку, могут быть назначены пользовательские атрибуты.

**Написание плагина**

Существует официальный сайт Tekla Structures, где содержится документация и разбор примеров по созданию плагинов: <https://developer.tekla.com/documentation/get-started-tekla-structures-open-api>

Там же находится ссылка на примеры плагинов на GitHub:

<https://github.com/TrimbleSolutionsCorporation/TSDirectManipulationPluginsExample>

Разбор примеров не очень глубокий и содержит автоматическое создание интерфейсов, из-за чего происходит связывание элементов, которые связывать не требуется. Мы же будем накладывать связи только на те элементы, которыми мы управляем.

**Пространства имен**

Написание плагина начинаем с названия - наш плагин будет называться Input\_Beam. Сразу создаем псевдонимы для пространств имен, которые будем использовать:

using TSP = Tekla.Structures.Plugins;

Включает в себя все функциональные возможности для создания подключаемого модуля в Tekla Structures.

using TSM = Tekla.Structures.Model;

Включает функциональность для вставки, выбора, изменения или удаления объектов внутри модели Tekla Structures. Пространство имен также включает возможности запрашивать различные данные из текущего экземпляра модели, например, информацию о выбранных в данный момент объектах.

using TSMUI = Tekla.Structures.Model.UI;

Пространство имен пользовательского интерфейса состоит из нескольких классов, которые можно использовать, например, для выделения объектов в пользовательском интерфейсе. В частности класс Picker, который может использоваться для запроса пользователя о выполнении ручного выбора объектов и точек из модели Tekla Structures.

using TSD = Tekla.Structures.Dialog;

Содержит классы и методы, которые позволяют, например, создавать диалоговые формы для взаимодействия с пользователем. Содержит классы для создания пользовательских интерфейсов как на WPF, так и на Windows Forms

using TSG = Tekla.Structures.Geometry3d;

Содержит необходимые базовые 3D-геометрические классы, которые используются Tekla Structures. Кроме того, предоставляются некоторые вспомогательные функции для упрощения использования этих классов.

using SWF = System.Windows.Forms;

Содержит классы для создания приложений Windows, которые позволяют наиболее эффективно использовать расширенные возможности пользовательского интерфейса, доступные в операционной системе Microsoft Windows.

using SD = System.Drawing;

Предоставляет доступ к основным графическим функциям GDI+.

**Атрибуты**

На плагин можно наложить дополнительные атрибуты, которые будут управлять его поведением. По умолчанию плагин является адаптивным, т.е. в случае изменений исходных данных, он будет пересчитываться. Но можно сделать такой плагин, который будет активироваться только один раз, т.е. его можно будет только вставить, после чего он больше не будет обновляться, что бы не происходило с исходными данными. Также могут быть дополнительные атрибуты, которые могут отвечать за дополнительные методы ввода. Перечень атрибутов небольшой, но они могут сделать поведение конкретного плагина особенным.

Чаще всего указываются два атрибута:

- Первый атрибут, отвечающий за имя плагина

- Второй атрибут отвечает за то, как Tekla будет реализовывать интерфейс

Остановимся на первом атрибуте:

[TSP.Plugin(nameof(Input\_Beam))]

Этот атрибут является обязательным, так как без него Tekla не будет знать, под каким именем хранить компонент в каталоге компонентов. Конструктор атрибута предполагает, что мы должны ввести string-значение в качестве аргумента. Предполагается, что имя компонента в каталоге может не совпадать с именем класса. Но для упрощения работы и поиска стоит присвоить одинаковые названия. Если вдруг мы поменяем название плагина, за счет используемой рефлексии, сразу же внутри алгоритма создастся новое имя.

В описании атрибута сначала идет псевдоним пространства имен(TSP), наименование атрибута(наиболее полное имя PluginAtribute, но суффикс Atribute можно не писать), а дальше идет конструктор атрибута, в котором мы должны ввести имя нашего плагина. Для того, чтобы сэкономить время, использыем рефлексию, т.е. мы в имени данного класса используем само наименование класса, используя метод nameof. Если вдруг мы неправильно напишем наименование класса, редактор сразу выдаст ошибку, что указан неверный класс (если класс не определен, nameof работать не будет).

Подытожим: этот атрибут нужен для того, чтобы Tekla распознавала данный плагин среди других, так как плагины друг от друга отличаются именами.

Второй атрибут:

[TSP.PluginUserInterface(nameof(Input\_Beam) + "." + nameof(Preset))]

Здесь указывается либо текст содержимого inp-файла, который передается Tekla, после чего она создает в специальной папке файл .inp с этим пользовательским интерфейсом, либо передается имя класса на Windows Forms или WPF, который Tekla будет использовать для генерации пользовательского графического интерфейса. И в данном случае мы указываем, что у нас есть пространство имен Input\_Beam и через "." наш класс интерфейса, который мы называем Preset.

**Класс Input\_Beam**

В пространстве имен Input\_Beam у нас будет класс Input\_Beam.

Есть несколко вариантов плагинов:

- ConnectionBase - первоначально создан для того, чтобы соединять между собой детали

- CustomPartBase - плагин для вставки отдельных сложных сущностей

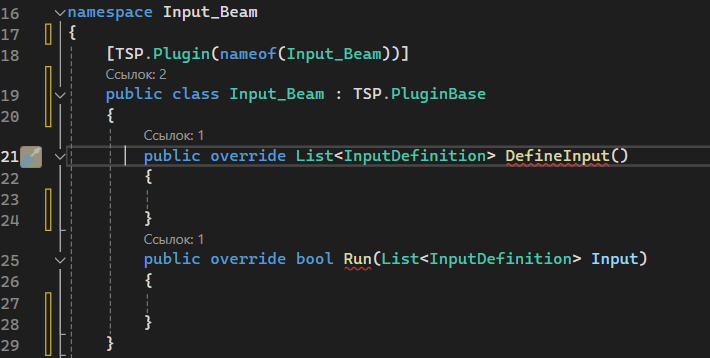
- DrawingPluginBase - плагин для создания объектов чертежа

- PluginBase - универсальный плагин

- и прочие

Класс Input\_Beam будет унаследован от PluginBase.

Программа сразу выдаст ошибку, что у нас не определены два метода, соглашаемся с предложением их реализовать.



**Метод DefineInput().**

Сигнатура метода:

- возвращает List<InputDefinition>

- без аргументов

Теперь необходимо определить DefineInput(). Именно в этой части мы и определяем, с чем будем работать. Для этого мы создаем класс Picker. Класс Picker может использоваться для запроса пользователя о выполнении ручного выбора объектов и точек из модели Tekla Structures. Класс содержит различные методы, вот некоторые из них:

- PickFace - Запрашивает у пользователя грани-плоскости на объекте модели

- PickLine - Запрашивает пользователя выбрать линию.

- PickObject - Запрашивает пользователя выбрать один объект модели.

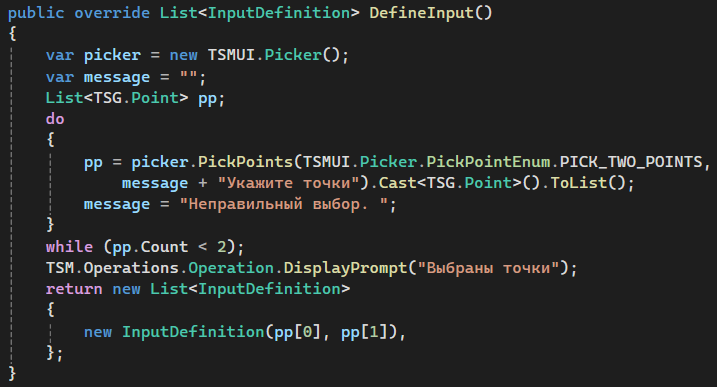
- PickPoint - Запрашивает пользователя выбрать точку на объекте модели.

Мы выбираем метод PickPoints, как так мы ставим балку по двум точкам, и определяем из встроенного перечислителя (PickPointEnum), какие именно точки мы собираемся выбирать. Мы выбираем две точки - PICK\_TWO\_POINTS и пишем сообщение для пользователя. Tekla использует абстрактные типы данных, поэтому мы будем вынуждены часто приводить тип данных от абстрактного к конкретному. Используя метод Cast приводим к типу данных List:

Cast<TSG.Point>().ToList()

Обернем picker в бесконечный цикл do-while, чтобы предотвратить неправильные действия пользователя (пользователь может не указать точки, указать только одну точку и так далее), в этом случае программа будет повторно просить у пользователя указать точки с помощью рекурсии. Чтобы убедиться, что метод отрабатывает, добавляем текст, который появится только после того, как точки выбраны.

Метод DefineInput возвращает список состоящий из двух точек, указанных пользователем. Мы создаем тип данных List<InputDefinition> и сразу же начинаем определять его, он состоит из одного единственного члена, в котором и хранятся две наши точки.



**Метод Run**

Сигнатура метода:

- возвращает void

- аргумент List<InputDefinition>

Переходим к переопределению метода Run**.** Внутри него используем try-catch, чтобы отловить возможные ошибки.

Вот краткий план, что именно необходимо сделать методу:

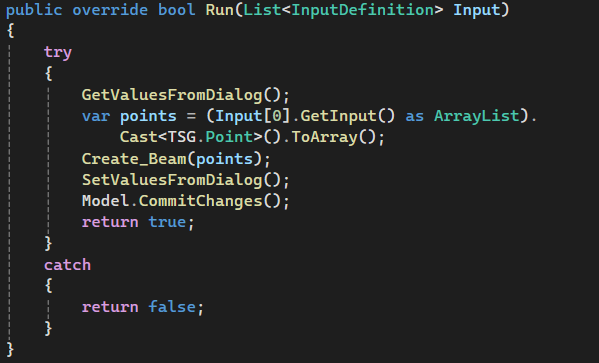
- получить атрибуты из пользовательского интерфейса(UI), используя метод GetValuesFromDialog

- извлечь данные, переданные через аргумент Input, используя метод GetInput и привести результат к массиву точек

- вызвать метод генерации балки - Create\_Beam

- назначить на созданный объект модели(плагин) атрибуты, которые были получены в пользовательском интерфейсе - SetValuesFromDialog

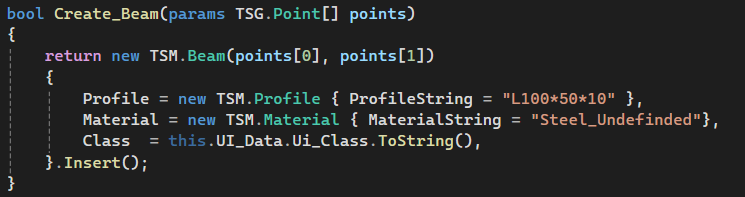
- обновить модель, принять изменения - Model.CommitChanges



Рассмотрим каждый этап подробнее.

В методе Run в первую очередь мы должны достать из аргумента метода Run те данные, которые мы в него вложили в методе DefineInput. Выходным типом данных в этом методе был List<InputDefinition> и именно его мы используем в качестве аргумента в методе Run, присвоив ему название Input. Мы берем этот аргумент и начинаем из него извлекать данные в переменную points. Input является индексированной коллекцией, но внутри нее может быть любой тип данных. В нашей коллекции содержится только один член, но их может быть любое количество. Мы обращаемся к этому элементу и чтобы мы могли им манипулировать дальше, мы применяем к нему метод GetInput (получить информацию,что находится внутри), в результате получаем некий object, который необходимо привести к ArrayList. Возможно оставить элемент к таком виде, но для дальнейшего использования удобнее при помощи метода Cast привести его к строго типизорованной коллекции - массиву точек.

Отдельно создаем метод для отрисовки балки Create\_Beam. В качестве аргумента используем params TSG.Point[] points.



Ключевое слово params означает, что мы должны ввести какой-то массив, но оно освобождает нас от необходимости в явном виде этот массив указывать, т.е. массив создается самим инструментом.

Задаем Profile (сечение балки) и Material (материал балки). То есть те значения, которые будут у балки по умолчанию, иначе балка не создастся. Всё это можно будет поменять через пользовательский интерфейс. Начальную и конечную точки мы указываем через аргументы конструктора (new TSM.Beam(points[0], points[1])) Также задаем цвет, который мы назначим в классе Ui\_Data, за цвет в Tekla отвечает свойство Class. Цвета задаются числом и в интерфейсе мы их тоже видим числом, но в системе они прописываются текстом. Поэтому цвет нужно обязательно привести к типу String.

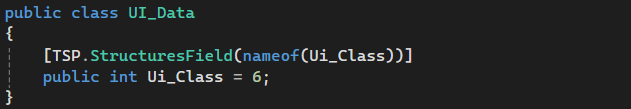
Возвращаемся к методу Run и вызываем метод Create\_Beam.

Теперь мы вызываем метод SetValueFromDialog(который будет описан позднее).

Используем метод CommitChanges, который фиксирует изменения, внесенные в базу данных модели на данный момент. Один коммит - это то, что пользователь может позже отменить с помощью команды undo. Коммит также запускает отображение измененной модели на видимых видах.

**Класс UI\_Data**

У плагина должен быть класс, в котором он хранит значение пользовательских переменных. Для этого нам необходимо создать публичный класс UI\_Data.



Это класс данных для интерфейса. Метод Run будет извлекать информацию определенную в пользовательском интерфейсе.

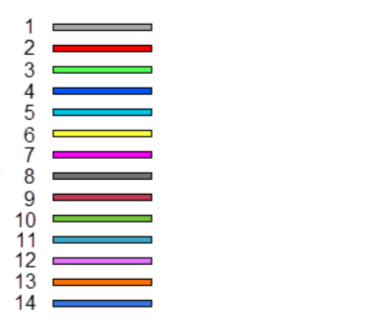
Плагин получает значения, которые ввел пользователь через значение членов этого класса. После того как плагин отработал, Tekla должна записать все значения этих атрибутов на экземпляр плагина в модели. Это нужно для того, чтобы при двойном щелчке мыши, Tekla сама извлекла ранее введенные значения. Чтобы Tekla понимала, что члены класса являются той информацией, которая должна попасть в интерфейс и которая должна записаться на экземпляр плагина в модели, нужно эту информацию пометить атрибутом. Атрибут называется - [TSP.StructuresField(nameof(Ui\_Class))]

Поле в этом классе должно иметь атрибут в пространстве имен TeklaStructuresPlugin под названием StructuresField и ему в качестве аргумента нужно записать его же, но мы так делать не будем, но для повышения прозрачности кода, мы будем использовать рефлексию через метод nameof. Атрибут StructuresField нужно обязательно назначать на те переменные, которые будут взаимодействовать с пользовательским интерфейсом, иначе они не будут работать.

При определении поля этого класса можно сразу указать дефолтные значения, которые в дальнейшем можно использовать. За счет этих дефолтных значений можно сэкономить много строчек кода. Здесь могут быть прописаны цвет, материал, профиль, длинна, угол поворота, также можно прописать координаты точек вставки в случае ошибочно указанных данных. Количество переменных определяются задачами, которые ставит перед собой плагин и сложностью проекта.

Члены интерфейсного класса могут быть только трех типов: double, int и string.

Создаем переменную для нашего интерфейса Ui\_Class, в ней будет храниться информация, например, о том, какого цвета будет наша балка, и присваиваем ей значение по умолчанию. Мы выбираем цвет 6 - желтый. Вот список остальных цветов по номерам:



Отметим еще раз, что пользовательский интерфейс напрямую с нашим плагином не взаимодействует, он взаимодействует только через объект класса UI\_Data.

**Класс Preset**

Создаем класс Preset, который и реализует пользовательский интерфейс,унаследованный от PluginFormBase(нужен для создания интерфейса для плагина):

public class Preset : TSD.PluginFormBase

У класса PluginFormBase родительским классом является класс Form из Windows Forms, поэтому его реализация не будет сильно отличаться от интерфейса обычных приложений на WinForms. Мы будем использовать табличную разметку, поэтому внутри мы будет элемент управления TableLayoutPanel. Элемент управления TableLayoutPanel упорядочивает содержимое в сетке. Так как макет строится как во время разработки, так и во время выполнения, его можно изменять динамически по мере изменения интерфейса плагина. Это позволяет пропорционально изменять размер элементов управления в панели, так чтобы они учитывали изменения, такие как изменение размера родительского элемента управления или длины текста в случае локализации.

Создаем элементы интерфейса:

- Main\_Panel - это разметочная таблица, в которую мы будем всё вставлять

- Ui\_Class\_CheckBox - CheckBox, который определяет учитывается или нет значение цвета, указанное пользователем, в перерасчетах плагина.

- TextBox - поле для ввода цифрового значения цвета

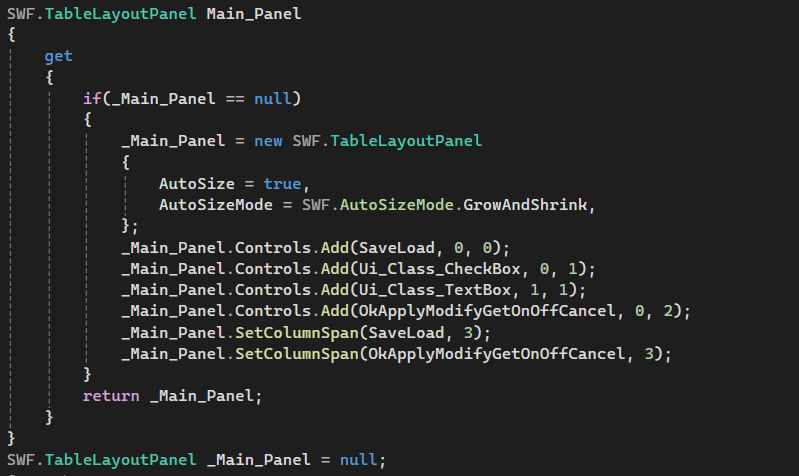
- OkApplyModifyGetOnOffCancel - стандартный блок клавиш управления плагином

- SaveLoad - блок сохранения или загрузки сохраненных значений.

Рассмотрим каждый из этих элементов.

**Main\_Panel**

В классе Preset создаем Main\_Panel класса TableLayoutPanel. Здесь мы определяем всего один аксессор - аксессор get. Это означает, что мы его можем только запросить, но можем принудительно создать. Также мы создаем невидимую (private) переменную (пишется через нижнее подчеркивание), которое по умолчанию будет иметь значение null. Т.е. по умолчанию оно не инициализировано и пустое. В аксессоре get создаем стандартную конструкцию: проверяем, что значение у \_Main\_Panel равно null, запускаем в этом случае инициализацию объекта и начинаем заполнять этот элемент управления другими элементами управления.



Создаем сам объект, используя его стандартный конструктор, и задаем ему значение AutoSize (задает значение, указывающее, производятся ли изменения размеров элемента управления в соответствии с его содержимым) - true. Можно было бы установить свой размер и стили, но мы этого делать не будем. Выбираем тип авторастяжения (AutoSizeMode), их существует два:

- GrowAndShrink. Элемент управления увеличивается или сжимается в соответствии со своим содержимым. Размер элемента управления нельзя изменить вручную.

- GrowOnly. Элемент управления увеличивается настолько, насколько это необходимо для соответствия его содержимому, но не уменьшается меньше значения его свойства Size. Размер формы можно изменять, но нельзя сделать ее настолько маленькой, чтобы какой-либо из содержащихся в ней элементов управления был скрыт.

Мы выбираем GrowAndShrink.

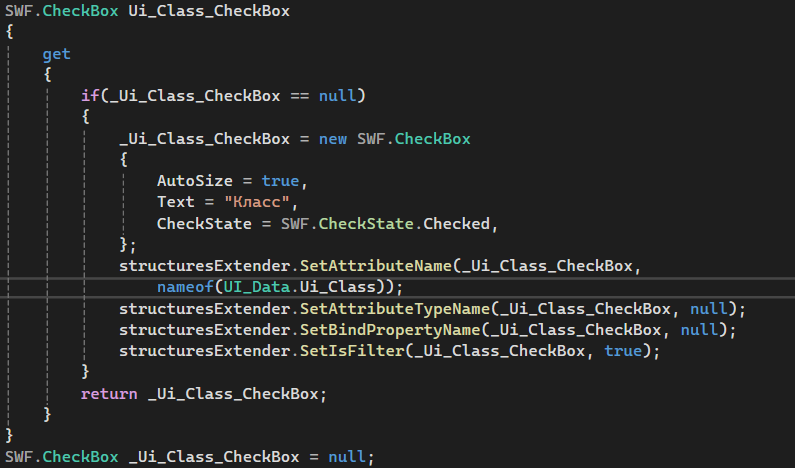
В конструкторе к классу Preset мы добавляем к члену Controls элемент управления Main\_panel.

Вставляем в таблицу созданные части интерфейса, каждую на определенное место. В скобках указываем название элемента и через запятую столбец и строчку, куда вставляется элемент. Далее используем SetColumnSpan, чтобы произошло объединение наших ячеек, чтобы интерфейс растянулся. SaveLoad и OkApplyModifyGetOnOffCancel растягиваем на 3 ячейки.

После чего мы завершаем блок инициализации. Далее возвращаем \_Main\_Panel. Т.е. в том случае, если он уже инициализирован, он сразу же будет запрашивать нашу приватную переменную. Все приватные поля этого класса мы создаем со значением null, пока алгоритм не дойдет до прямого выполнения свойств, элементы создаваться не будут.

**Ui\_Class\_CheckBox**

Переходим к элементу Ui\_Class\_CheckBox.



Делаем ему также AutoSize и пишем текст, где указываем его псевдоним (отображаемое имя в интерфейсе пользователя).

Критически важно указать CheckState (возвращает или задает состояние CheckBox) в значении true. Без этого пользовательский интерфейс работать не будет. Мы устанавливаем значение Checked = true.

Далее нам нужно связать элементы интерфейса, помогая понять форме PluginFormBase, то что данный элемент интерфейса должен провзаимодействовать с назначенным элементом из Ui\_Data.

Для этого используем класс StructuresExtender. Класс StructuresExtender предназначен для привязки элементов управления к типам данных. Привязки необходимы для передачи значений диалогового окна в Tekla Structures и далее в подключаемые модули.

Мы будем использовать следующие его методы:

- SetAttributeName - связывает имя атрибута с указанным элементом управления

- SetAttributeTypeName - связывает тип атрибута с указанным элементом управления.

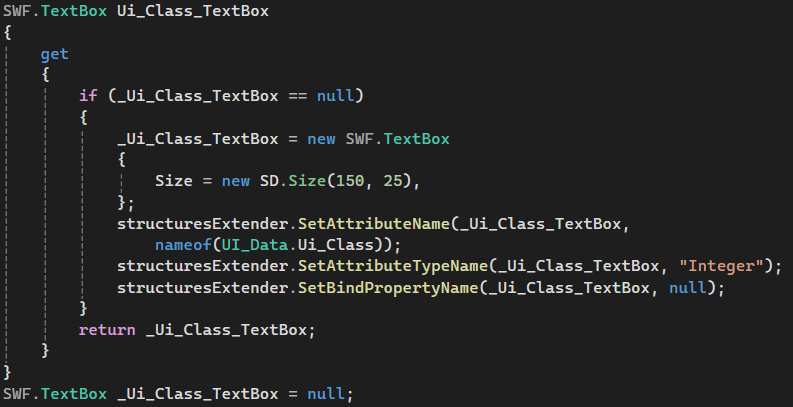
- SetBindPropertyName - связывает тип извлекаемой информации с указанным элементом управления.

- SetIsFilter - связывает логическое значение IsFilter(является ли фильтром) с указанным элементом управления. Указывается только для CheckBox.

**TextBox.**

Далее переходим к полю для ввода TextBox. Оно не может быть AutoSize, поэтому ему необходимо установить размер с помощью SystemDrawing.Size. 150 на 25 пикселей - оптимальный размер для этого поля.

Так же как мы привязывали CheckBox, привязываем поле для ввода при помощи structuresExtender.



В примерах на сайте Tekla помимо TextBox и CheckBox предлагается привязать MainPanel. Однако, это действие не является обязательным, так как выставляется значение null и не несет никаких изменений в плагине.

**OkApplyModifyGetOnOffCancel и SaveLoad**

Для создания полноценной формы нам необходимо создать верхнюю и нижнюю части этой формы. Для этого мы обращаемся к пространству имен Tekla.Structures.Dialog.UIControls, которое содержит шаблоны диалоговых окон и пользовательские элементы управления.

Нам понадобятся класс OkApplyModifyGetOnOffCancel, который представляет собой элемент управления, включающий группу кнопок Ok-Apply-Modify-Get-On/Off-Cancel.

А также класс SaveLoad, который представляет группу элементов управления save-load-save as.

Начинаем с класса OkApplyModifyGetOnOffCancel. Этот элемент будет автоматически масштабироваться по своим габаритам. Но при этом обязательно устанавливаем минимальный размер. Дальнейший код класса является стандартным и может быть скопирован без изменений для другого проекта.

Список событий и кнопок, за которые они отвечают:

- Событие ApplyClicked вызывается при нажатии кнопки Apply.

- Событие CanselClicked возникает при нажатии кнопки Cansel.

- Событие GetClicked возникает при нажатии кнопки Get.

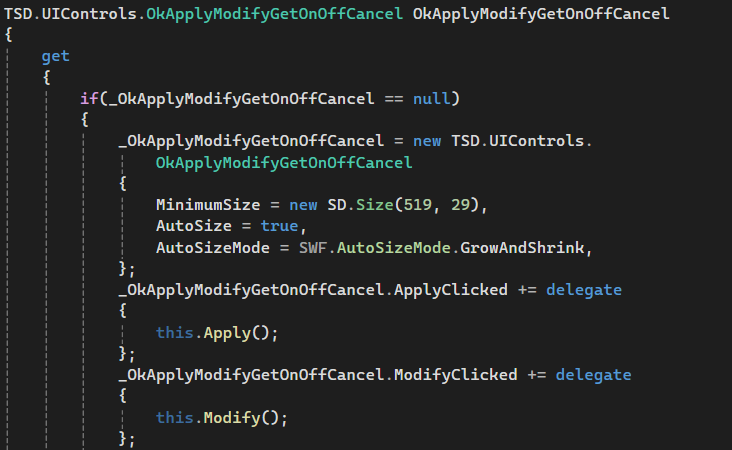
- Событие ModifyClicked вызывается при нажатии кнопки Modify.

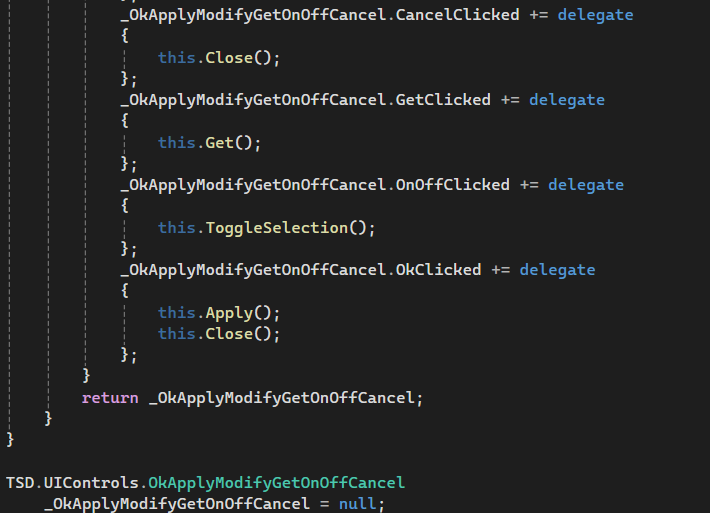
- Событие OkClicked вызывается при нажатии кнопки Ok.

- Событие OnOffClicked вызывается при нажатии кнопки On-Off.

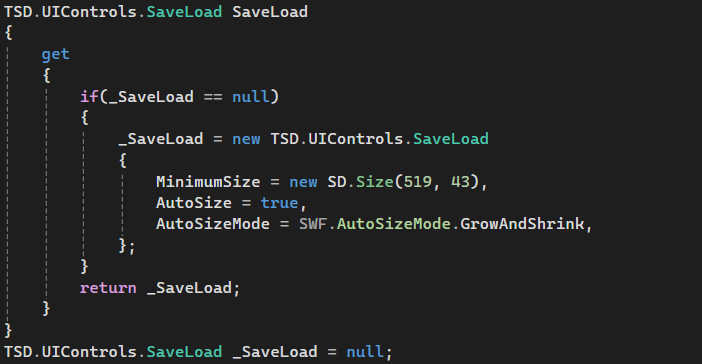
У нас есть блок с вышеперечисленными кнопками, но на эти кнопки не привязаны никакие функции, но эти функции находятся в классе формы - PluginFormBase (является базовым классом для диалоговых окон плагина, класс расширяет базовый класс Form, добавляя связи со структурами Tekla) и мы связываем встроенные функции класса формы с кнопками этого блока.

Для каждого из них добавляем обработчик через анонимный метод, используя делегат. Можно было бы использовать делегат класса EventHandler(предопределенный делегат, который представляет метод обработчика событий для события, которое не создает данные). Но мы используем упрощенный синтаксис, так как мы не передаем в отработчик никаких атрибутов и ничего не считываем. У нас есть блок с кнопками, внутри блока есть события (нажатие определенной клавиши) и мы на это событие накладываем обработчик события, описываемый через делегат. В делегате мы не указываем атрибут, так как не предполагаем, что мы что-то дополнительно сюда передаем. Теперь при нажатии на соответствующие кнопки будет происходить определенное событие, иначе говоря кнопки становятся рабочими.

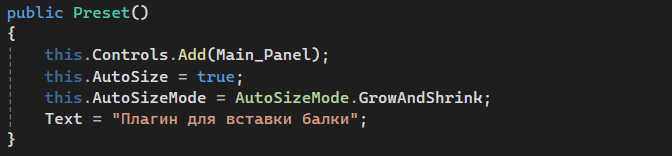




Переходим к классу SaveLoad. Выполняем аналогичные действия, которые проделывали с предыдущим классом (кроме привязки кнопок). Этот блок также стандартный и его можно копировать из проекта в проект.



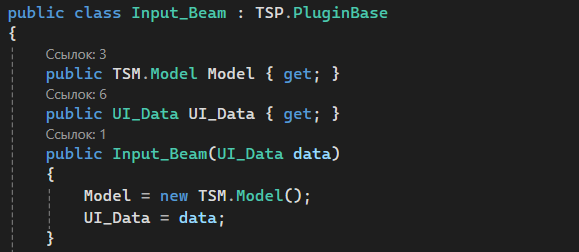
Пишем конструктор класса Preset. Он должен быть публичным и не содержать никаких аргументов.



Здесь мы добавляем в Controls Main\_Panel, т.е. в форму для вывода мы добавляем элемент управления - нашу табличную разметку, которая подтягивает за собой кнопки, блоки и т.д., то есть всё то, что мы добавили в элементы управления. Мы используем табличную форму управления, так как она является самой удобной, она автомасштабируется и нам не нужно работать с координатами элементов.

Так как наша форма для вывода имеет заголовок, далее пишем название плагина. Мы можем через рефлексию указывать техническое наименование плагина или использовать статический текст.

Далее мы объявляем члены класса Input\_Beam и конструктор



Мы объявляем автосвойства Model и UI\_Data. Мы подразумеваем, что они могут быть заданы принудительно только в конструкторе, а дальше они могут только читаться. А приватные переменные создаются уже компилятором, таким образом экономятся строчки кода.

Создаем стандартный консруктор Input\_Beam, который использует в качестве аргумента UI\_Data. Т.е. мы передаем полученное значение интерфейса в сам плагин.

Нам нужно добавить еще два важных метода.

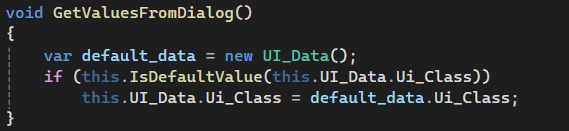
**Метод GetValuesFromDialog()**

Сигнатура:

- возвращает void

- без аргументов

Первый метод - GetValuesFromDialog. Название не отображает суть метода, но именно в таком названии он существует в примерах для написания плагинов. Наш метод отличается от приведенных в интернете примеров и его можно использовать в таком виде в своем проекте.



Компонент не умеет хранить дефолтные значения, вместо них он хранит крайние значения переменной. То есть в случае с числом он хранит самое минимальное значение, которое может быть, по мнению Tekla (внутри себя Tekla может оперировать другими типами данных, для этого существует специальная библиотека DataTypes). Если экземпляр плагина имеет какое-то крайнее значение, это проверяется методом IsDefaultValue, то данное значение признается дефолтным и оно этим методом заменяется на привычное нам дефолтное значение. В начале метода мы создаем новый экземпляр класса UI\_Data. Этот экземпляр создается только в пределах данного метода и на него не влияет пользовательский интерфейс. Он нам нужен извлечения дефолтных значений. Далее через условный оператор if мы проверяем, дефолтное ли это значение, и если да, то ему присваивается значение нашего дефолтного значения, которое находится в UI\_Data.

Этот метод нужно вызывать в начале метода Run, так как его нужно выполнить прежде чем мы начинаем свой алгоритм.

**Методт SetValuesFromData**

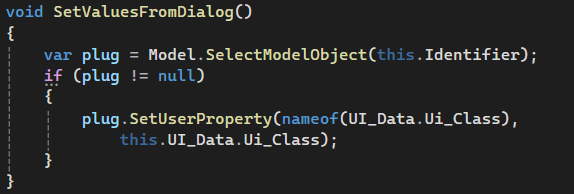
Сигнатура:

- возвращает void

- без аргументов

Теперь нам необходимо сделать второй метод - SetValuesFromData, он будет записывать данные из интерфейса на объект модели, который мы создаем.

Для того, чтобы получить экземпляр плагина в модели используем метод SelectModelObject. Он принимает в качестве входных данных идентификатор объекта. Затем проверяем, что метод не вернул пустое значение. И на данный объект модели назначаем пользовательский атрибут через метод SetUserProperty. Мы используем условный оператов if, чтобы во время отладки не возникало ошибок. Этот метод необходимо корректировать в случае изменения количества членов класса UI\_Data.



Вызываем этот метод

**Метод Main**

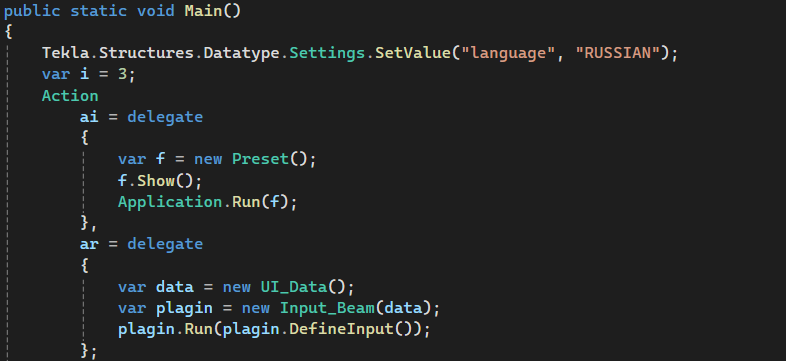
Сигнатура:

- возвращает void

- без аргументов

Необязательной частью плагина является метод для отладки. Без него не сможем тестировать наш плагин в Visual Studio.

Создаем для этой цели метод Main (делаем его публичным и статичным). Именно здесь мы будем проверять, что наши алгоритм и интерфейс функционируют правильно.



В первой строчке мы определяем с какими настройками будет запускаться интерфейс ("language","RUSSIAN"). Когда мы запускаем плагин через Tekla, она сама накладывает эти характеристики. Когда мы его запускаем по сути в качестве отдельно стоящего приложения, мы должны это прописать сами.

Есть несколько вариантов запуска отладчика:

- сначала создаем и показываем интерфейс, а потом запускаем алгоритм

- запускаем алгоритм, а потом показываем интерфейс

- только алгоритм

- только интерфейс.

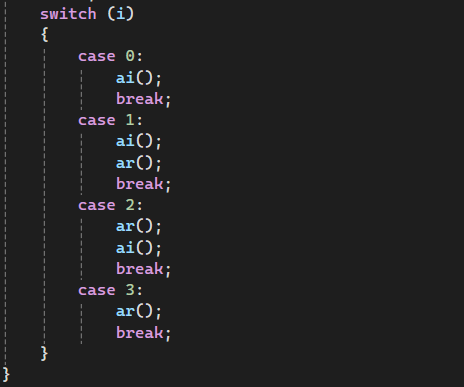
Чтобы все это не писать по несколько раз, мы эти типовые действия описываем в качестве делегатов, а потом эти делегаты запускаем в случае определенного значения. Сначала мы определяем какие именно действия. Но исполняться они будут в другом месте.

Внутри делегата мы создаем объект класса Preset, применяем к нему метод show, используем класс Aplication внутри Windows Forms и запускаем приложение с помощью нашей формы.

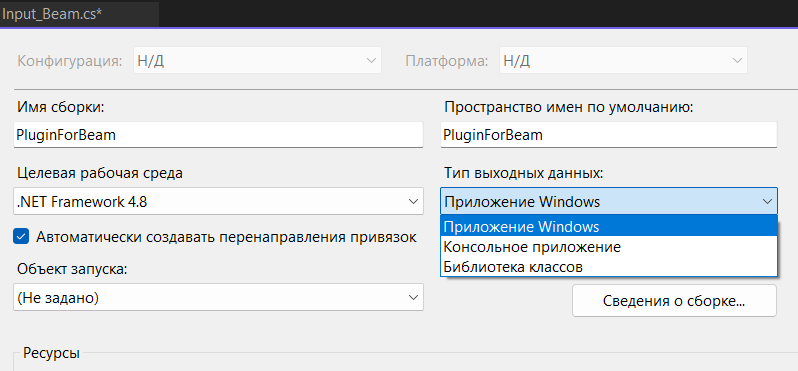
Во второй части создаем экземпляр нашего плагина,

Далее идут способы запуска.

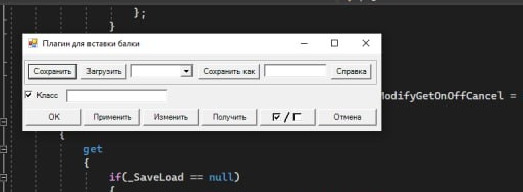
В делегатах мы только определили действия, а далее через switch определяем, что именно будет запускаться: только интерфейс, или только алгоритм, или все вместе (записаться они могут в разном порядке). Теперь мы сможем задавать, что именно мы отлаживаем.



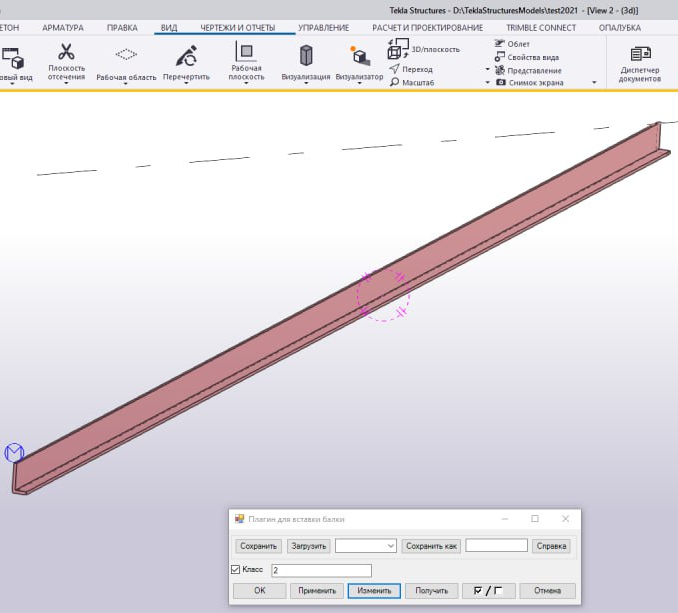
Для отладки нам нужно перейти в свойства проекта и выбрать "Тип входных данный" - "Приложение Windows", так как отлаживать мы его можем именно в режиме приложения.



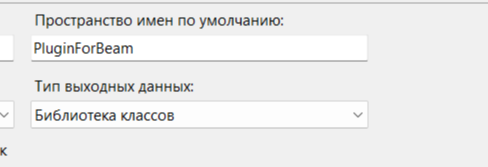
В результате проверки интерфейса (выбираем case 0) убеждаемся, что он отображается верно. Получаем следующее окно



Далее проверяем сам алгоритм (case 3). Указываем на модели две точки и видим, что балка строится



**Чтобы плагин заработал**, изменяем в свойствах проекта "Тип выходных данных" обратно на "Библиотека классов"



Теперь проект нужно "Пересобрать" или "Собрать", нажав правой кнопкой мыши на название плагина. Далее также, нажав на название плагина, выбираем "Открыть папку в проводнике" => bin => Debug, здесь и находится наш плагин с расширением .dll.

Далее находим, где у нас установлена Tekla - папка TeklaStructures => выбираем версию программы => nt => bin => plugins и можно либо сюда поместить файл, либо создать свою папку.

Теперь запускаем клиент Tekla, запускаем любую модель (после перемещения в нужную папку, плагин доступен в любом проекте).

Важно понимать, что изменение имени плагина, означает создание нового плагина, все предыдущие вхождения плагина в программу будут неработоспособными (т.е. созданные плагином объекты продолжат существовать, но если плагин предполагает дальнейшую работу с элементами, этого делать не получится, с элементом можно будет продолжить только в рамках инструментария самой Tekla).

В случае многопользовательской модели, если разные пользователи имеют разные плагины имеющие одинаковое имя, у каждого из них экземпляр плагина будет работать по алгоритму его локального плагина.

В Tekla можно проверить, загрузился ли наш плагин. Для этого заходим в "Журналы"=>"Журнал истории сеанса", здесь показывается вся загрузка Tekla и здесь можно найти и наш плагин.

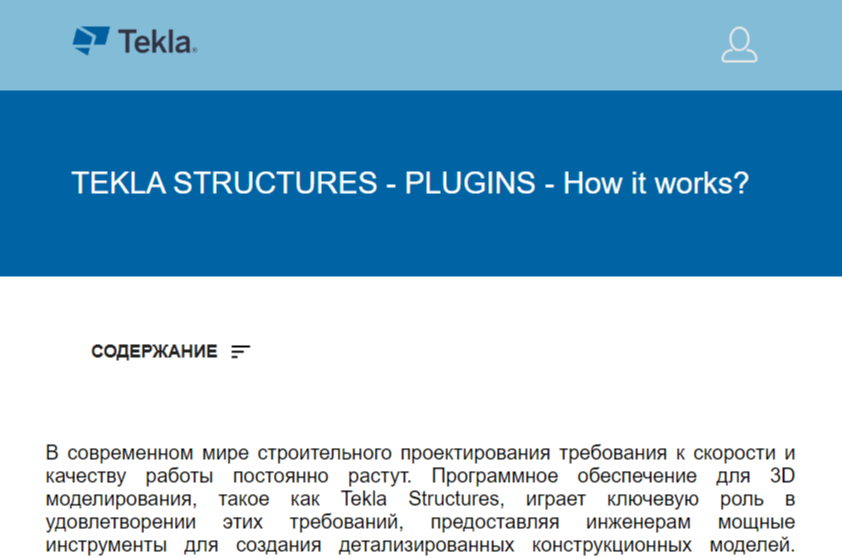
Мы сделали плагин для отрисовки простой балки по двум точкам, но мы могли сделать так, чтобы рисовались еще какие-то балки, чтобы они сваривались между собой, чтобы двумя кликами мыши рисовалась целая сложная конструкция.

**Сайт.**

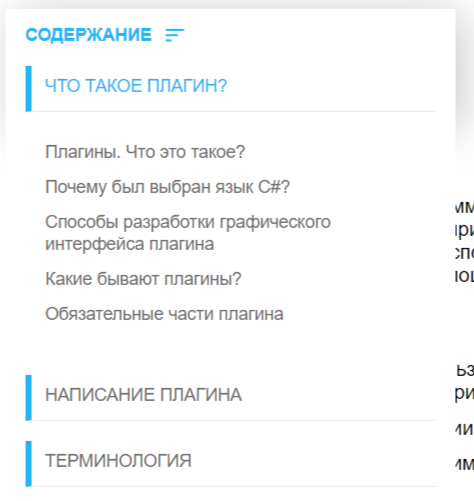
В настоящий момент сайт по данному руководству находится в разработке.

Он большей частью копирует информацию из диплома по созданию плагина. В настоящий момент сайт содержит:

Главную страницу:



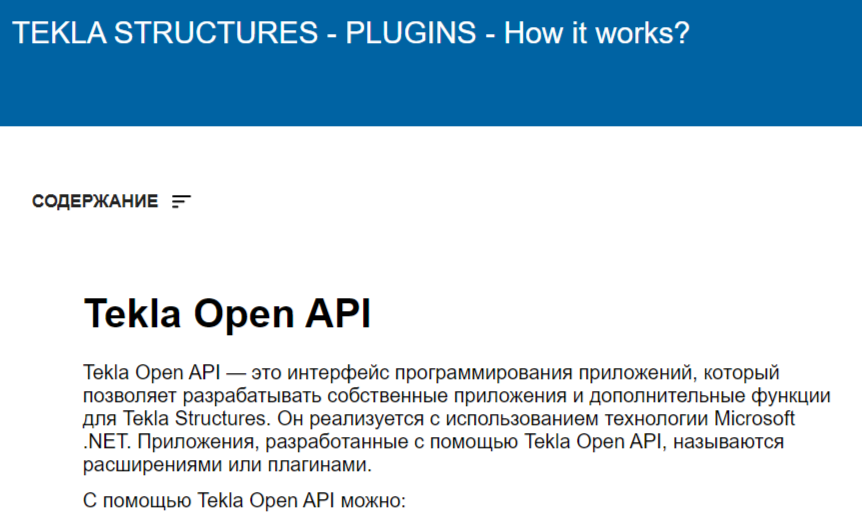
Содержание:



Страницы по созданию плагина:



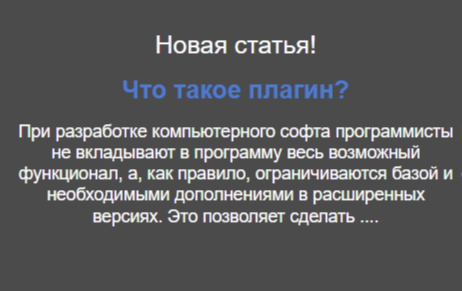
Страницы по терминологии:



С главной страницы, щелкнув по значку Tekla можно перейти на сайт-руководство разработчиков Tekla.

В дальнейшем работа над сайтом будет продолжена. В частности будут добавлены разделы по созданию более сложных плагинов, появится возможность комментрировать статьи после регистрации. Соответственно появится личный кабинет пользователя. Сейчас в верхнем правом углу присутствует иконка пользователя, но регистрация пока недоступна. В настоящий момент иконка служит для перехода на главную страницу с остальных страниц.

В нижней части главной страницы будет появляться информация о вновь вышедших статьях.



В самой нижней части страницы будут различные ссылки: на сайты по теме, телеграмм-канал, социальные сети.

Сайт уже сейчас может служить хорошим руководством для начинающих разработчиков плагинов для Tekla Structures c использованием Tekla Open API.

**Заключение**

Руководство будет размещено на отдельном сайте, который пока находится на стадии разработки. К работе приложена ссылка на GitHub, где размещен проект сайта.

В данной работе мы рассмотрели лишь самый простой вариант плагина. Используя наш шаблон, пользователь может создать свой несложный плагин. Безусловно, для написания более сложных требуется дальнейшая разработка руководства.

Так как руководство представляют большую ценность для пользователей, работа над сайтом также будет продолжена.

В ходе выполнения дипломной работы была разработана и представлена методика создания плагинов для программы Tekla Structures с использованием Tekla Open API на языке C#. Результаты исследования показали, что разработка пользовательских плагинов значительно повышает эффективность проектирования строительных конструкций, позволяя инженерам адаптировать программное обеспечение под специфические задачи проекта.

Созданное руководство служит важным ресурсом для инженеров и разработчиков, желающих расширить функциональность Tekla Structures и оптимизировать рабочие процессы. Рекомендации и примеры кода, представленные в работе, могут быть использованы в качестве основы для дальнейшего развития и адаптации инструментов моделирования.

В заключении следует отметить, что потенциал использования Tekla Open API весьма значителен и его возможности далеко не исчерпаны. Перспективы дальнейших исследований могут включать разработку более сложных плагинов, интеграцию с другими программными решениями и автоматизацию процессов, что открывает новые горизонты для инноваций в строительной индустрии.

**Список литературы и ресурсов:** 1. Васильева А.Н. Программирование для начинающих. Москва 2021 г.

2. Герберт Шилдт. С# 4.0. Полное руководство. Москва 2020 г.

3. <https://ru.wikipedia.org/wiki/Tekla_Structures>

4. <https://practicum.yandex.ru/blog/chto-takoe-api/>

5. <https://opensourcebiology.eu/>

6. <https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/>

7. <https://ru.wikipedia.org/wiki/Плагин>

8. <https://developer.tekla.com/documentation/get-started-tekla-structures-open-api>

9. <https://www.rush-analytics.ru/blog/chto-takoe-plagin-dlya-sajta-zachem-nuzhen-i-chto-on-oznachaet-opredelenie-ponyatiya-prostymi-slovami>

10. <https://ru.wikipedia.org/wiki/Windows_Forms>

11. <https://ru.wikipedia.org/wiki/Windows_Presentation_Foundation>

12. <https://support.tekla.com/ru/doc/tekla-structures/2021/det_modify_cc_input_file_manually>