Разработка проверок в примерах

1. Введение

Данная функция является экспериментальной и доступна только на специальных сборках 1C:EDT (https://edt.1c.ru/releases/experimental/validation)

Новый механизм автоматической проверки конфигураций 1C:EDT значительно расширяет возможности пользователей как по управлению процессом проверки, так и по разработке собственных проверок. Целью данного документа является поэтапное введение пользователей в возможности данного механизма, а также демонстрирует процесс разработки новых расширений в виде развернутого набора примеров.

1.1. Основные понятия

Новый механизм основывается на понятии **"проверки"** - отдельного компонента, содержащего как, собственно, логику проверки того или иного сценария/набора данных, так и весь необходимый набор дополнительной информации, позволяющей данной проверке эффективно выполняться в процессе внесения изменений в конфигурацию пользователем или автоматизированными процессами разработки. Кроме непосредственно *проверки*, существует ряд терминов, элементов и процессов решения, упоминание которых встречается в данном документе:

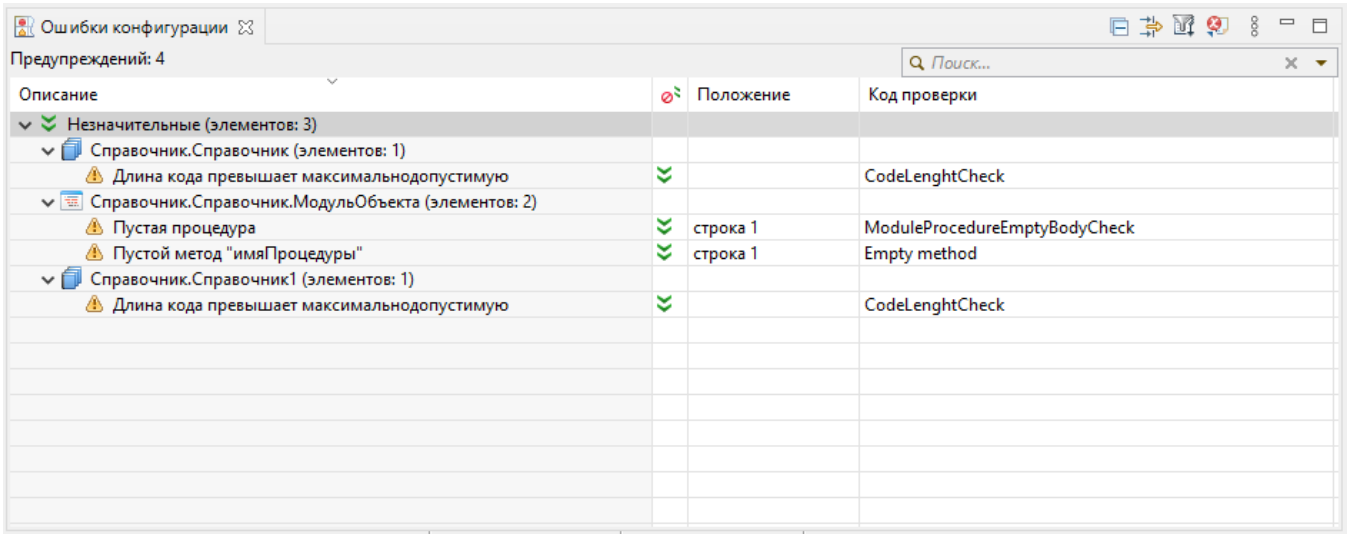
- **Проверка (Check)** - собственно, проверка
- **Проблема (Issue)** - некоторая ситуация (обычно - проблемная) в данных конфигурации, требующая внимания разработчика данной конфигурации
- **Маркер (Marker)** - результат работы проверки в случае обнаружения целевой проблемы (для проверки) в проверяемых данных. Маркеры отображаются в соответствующей панели 1C:EDT и содержат всю необходимую (по мнению автора проверки) информацию для нахождения и последующего исправления целевой проблемы
- **Код проверки** (**Check Identifier**)- уникальный идентификатор проверки, который может быть использован для управления настройками данной проверки, фильтрации маркеров, созданных данной проверкой и т.п.
- **Параметры проверки** - опциональный набор параметров, настраиваемых пользователем в профиле настроек для каждой параметризованной проверки. Введенные значения параметров могут использоваться логикой проверки, настраивая ее поведение в соответствии с целями пользователя проверки
- **Дескриптор проверки** - данные, предоставляемые разработчиком проверки, описывающие ее поведение в случае изменения данных, набор допустимых параметров, уникальный код проверки и т.п. Дескрипторы проверки используются 1C:EDT для встраивания проверки в механизм проверок и обеспечения ее жизненного цикла
- **Языковые проверки** - проверки, выполняемые на языковых модулях. Обладают определенной спецификой (поскольку определенной спецификой обладают и сами модули, по отношению к объектам конфигурации)
- **Объектные проверки** - проверки, выполняемые на объектах конфигурации (объектах метаданных, внешних свойствах и т.п.)

1.2. Интерфейс работы с проверками

Зафиксировав основные понятия подсистемы проверок 1C:EDT, сразу же перейдем к ее пользовательскому интерфейсу.

1.2.1. Панель "Ошибки конфигурации"

Все результаты работы проверок можно найти в панели 1C:EDT "Ошибки конфигурации":



Для каждой обнаруженной проверками проблемы, 1С:EDT создает маркер, и именно эти маркеры мы и видим в таблице проблем. Каждый маркер содержит:

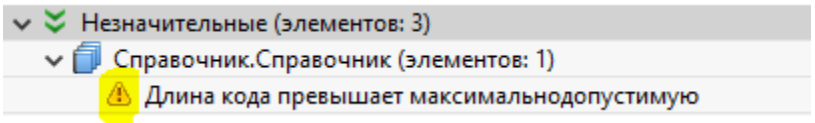
- Тип проблемы
- Критичность проблемы
- Описание обнаруженной проблемы
- Положение в тексте (если применимо)
- Код проверки, выявившей проблему
- Неявная ссылка на целевой объект, в котором проверка и обнаружила данную проблему

Остановимся подробнее на специфике каждого из атрибутов маркера.

1С:EDT поддерживает следующие **типы** проблем:

- Ошибка (Error)
- Предупреждение (Warning),
- Безопасность (Security),
- Производительность (Performance),
- Переносимость (Portability)
- Разработка и использование библиотек (Library development and usage)
- Стандарты кодирования (Code style)
- Стандарты разработки интерфейсов (UI style)
- Орфография (Spelling

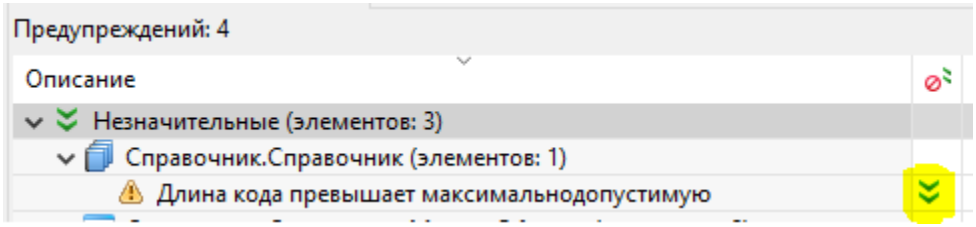
Тип проблемы отображается в виде соответствующей иконки рядом с описанием:



Критичность проблемы определяет ее влияние на процесс разработки и конечный продукт, и может иметь следующие значения:

- Блокирующая (Blocker)
- Критическая (Critical)
- Значительная (Major)
- Незначительная (Minor)
- Тривиальная (Trivial)

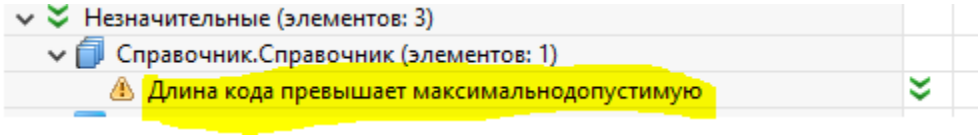
Критичность проблемы отображается в соответствующей колонке таблицы:



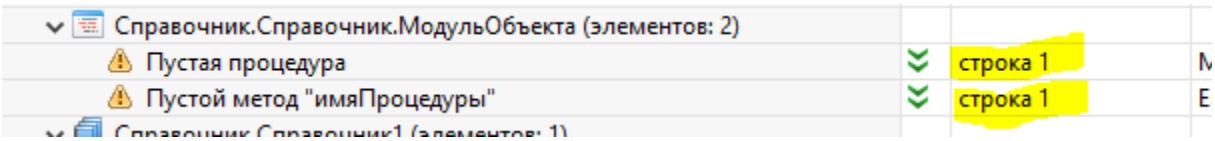
Кроме того, все маркеры сгруппированы по трем группам критичности, отсортированными по приоритету исправления соответствующей проблемы:

- *Документ в разработке*

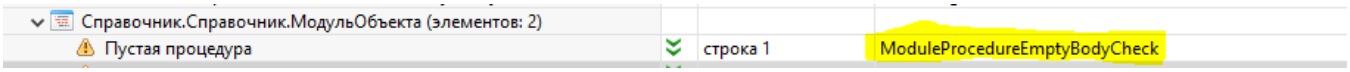
Описание обнаруженной проблемы полностью зависит от логики проверки, и представляет собой произвольный текст:



Положение в тексте применимо только для маркеров, созданных в результате проверок языковых модулей конфигурации. В зависимости от ситуации, может быть сохранена как строка, так и колонка, соответствующая месту возникновения проблемы:



Код проверки всегда соответствует той проверке, которая создала данный маркер. Код позволяет находить все маркеры, созданные данной проверкой, а также оперативно менять настройки проверки (вплоть до отключения или подавления ошибок):



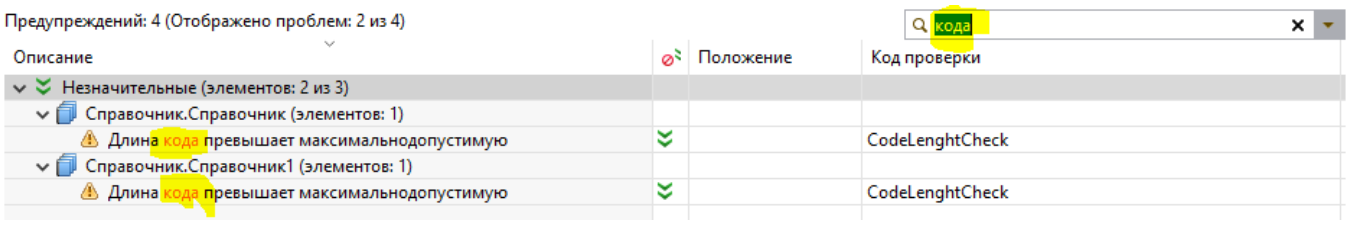
1.2.2. Поиск и фильтрация проблем

Теперь, ознакомившись с основными атрибутами маркеров проблем, познакомимся со средствами поиска и фильтрации в окне ошибок конфигурации.

Начнем с **поиска** (на самом деле - фильтрации) маркеров по тексту. Механизм поиска ищет совпадения в:

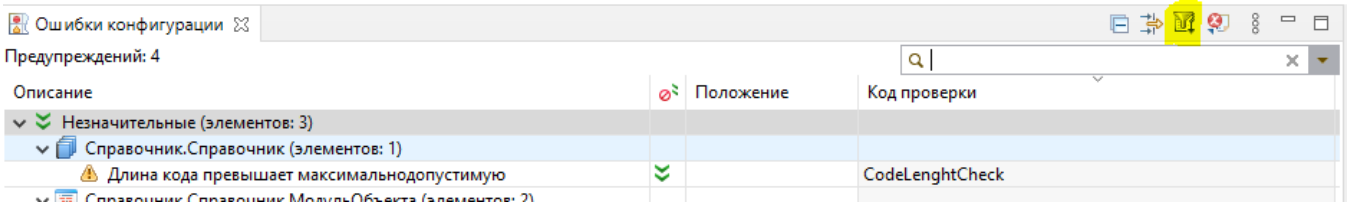
- Имени объекта
- Описании проблемы
- Коде проверки
- И имени объекта/модуля, в котором была обнаружена проблема

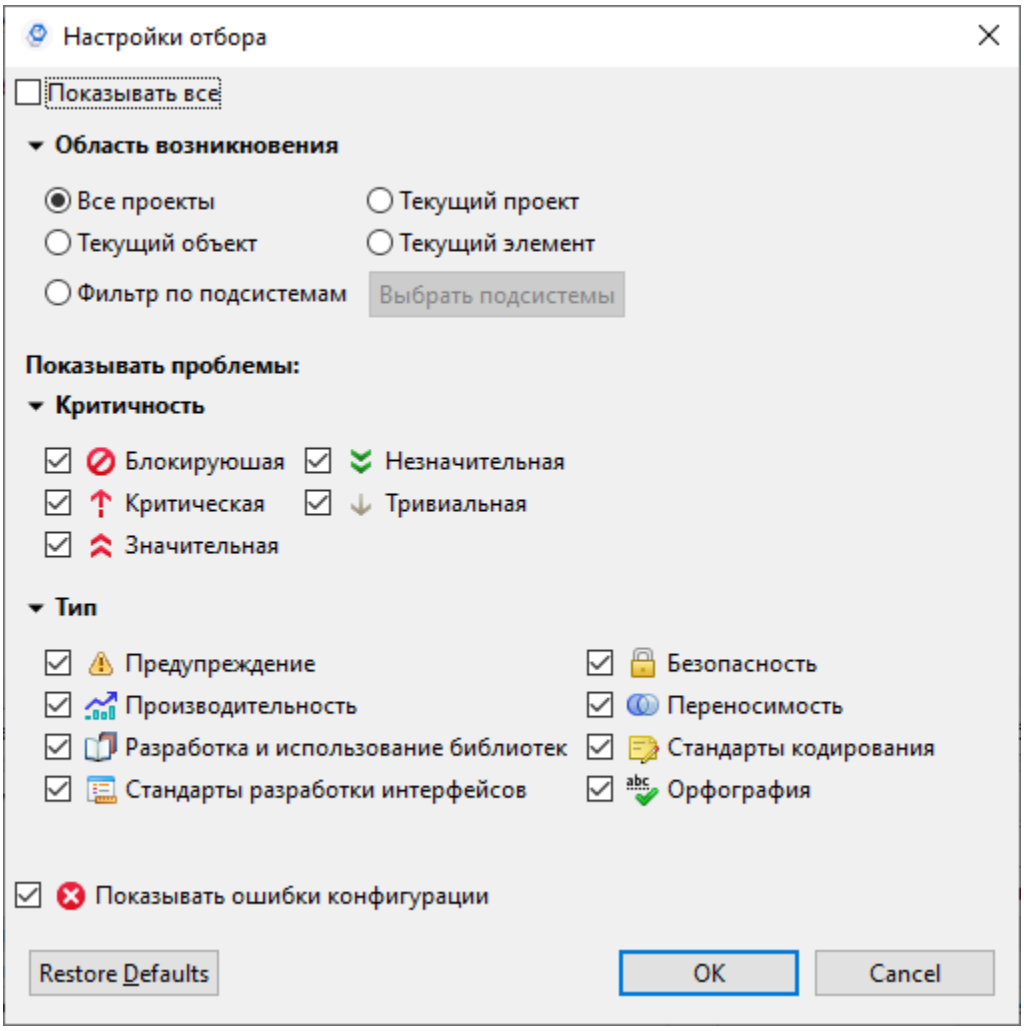
Поиск производится по подстроке, указания wildcard-символов не требуется. Результатом поиска является отфильтрованная по совпадениям таблица:



Текущий поиск можно отменить, нажав крестик в поле поиска. 1С:EDT сохраняет историю поиска, поэтому можно выбирать предыдущие поисковые строки из выпадающего списка поля поиска. Смена сортировки в списке, переход к месту проблемы и т.п. не сбрасывает текущую строку поиска.

Строка поиска используется для оперативного поиска единичных проблем, либо для поиска определенных проблем по их тексту/коду проверки. Для сложного анализа/отбора проблем используется механизм фильтрации отображаемых проблем, диалог настроек которого вызывается при нажатии на соответствующую кнопку в панели инструментов окна ошибок конфигурации:





Рассмотрим возможности фильтрации маркеров проблем в деталях.

Область возникновения - задает фильтрацию отображаемых маркеров в зависимости от текущего выбранного в навигаторе /редакторе объекта/ов. Допускаются следующие варианты фильтрации по области:

- Все проекты (значение по умолчанию) - будут показаны все маркеры, без фильтрации по выбранному объекту и/или проекту
- Текущий проект - будут показаны все маркеры, относящиеся к проблемам в выбранном в данный момент проекте. Выбор проект определяется активным выбором в дереве навигации и/или активном редакторе (в случае, если настроена автоматическая установка выбора в навигаторе при выборе редактора)
- Текущий объект - будут показаны все маркеры, относящиеся к проблемам в выбранном в данный момент объекте, а также всех его зависимых объектах (формах и т.п., если такие присутствуют)

- Текущий элемент
- Фильтр по подсистемам - отбор маркеров по составу выбранных подсистем

Остановимся подробнее на отборе маркеров по составу выбранных подсистем. Функция, в общем-то, является стандартной для многих механизмов 1С:EDT и платформы. Для установки фильтра по подсистемам необходимо нажать кнопку "Выбрать подсистемы", в результате чего на экране появится диалог фильтрации по подсистемам:

- *Документ в разработке*

Показывать проблемы по критичности - фильтрация маркеров по критичности соответствующей проблемы. Позволяет выбрать любой произвольный набор из доступных критичностей проблем.

Показывать проблемы по типу - полностью аналогично фильтрации по критичности по поведению. Позволяет выбрать любой произвольный набор типов проблем для отображаемых маркеров.

Показывать ошибки конфигурации

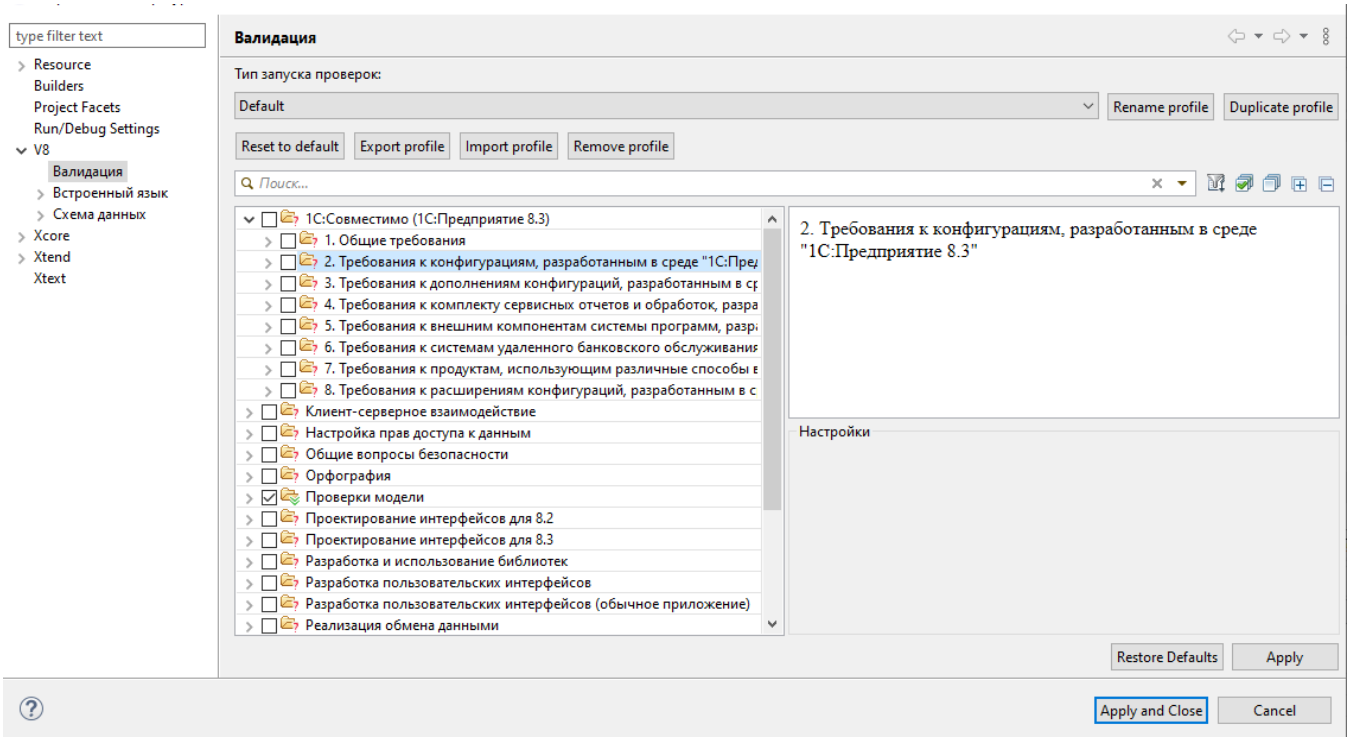
1.2.3. Переход к источнику возникновения проблемы

Для эффективной работы с обнаруженными проблемами, панель ошибок конфигурации позволяет производить переход к конкретному месту возникновения проблемы. Для этого необходимо выбрать соответствующий маркер проблемы (строку в таблице проблем), и дважды нажать левой кнопкой мыши. 1С:EDT автоматически откроет необходимый редактор и перейдет к (при наличии такой возможности) элементу, связанному с проблемой.

1.2.4. Настройки проверок

Пользователь 1С:EDT имеет возможность управления настройками проверок, адаптируя их работу к особенностям бизнес-процесса разработки конкретной команды или проекта. Все настройки проверок хранятся на уровне проекта, обеспечивая их переносимость при командной работе, портировании проектов и т.п. операций.

Настройки проверок доступны через окно настроек валидации конфигурации: **ПКМ на элементах проекта в навигаторе - Свойства - V8 - Валидация**:



Остановимся на основных моментах при настройке поведения проверок.

1.2.4.1. Профили настроек

Для обеспечения возможности многопрофильной проверки (например - на различных фазах разработки), 1С:EDT поддерживает произвольный именованный набор измененных настроек проверок - так называемый профиль настроек.

Список текущих профилей настроек, а также элементы управления находятся в верхней части страницы настроек:

Пользователь может:

- Выбирать активный профиль настроек из списка существующих
- Переименовывать выбранный профиль
- Дублировать выбранный профиль, с заданием имени для вновь создаваемого профиля
- Сброс настроек профиля до настроек по умолчанию для каждой проверки
- Экспорт профиля во внешний файл
- Импорт ранее экспортированного профиля из внешнего файла
- Удаление существующего профиля

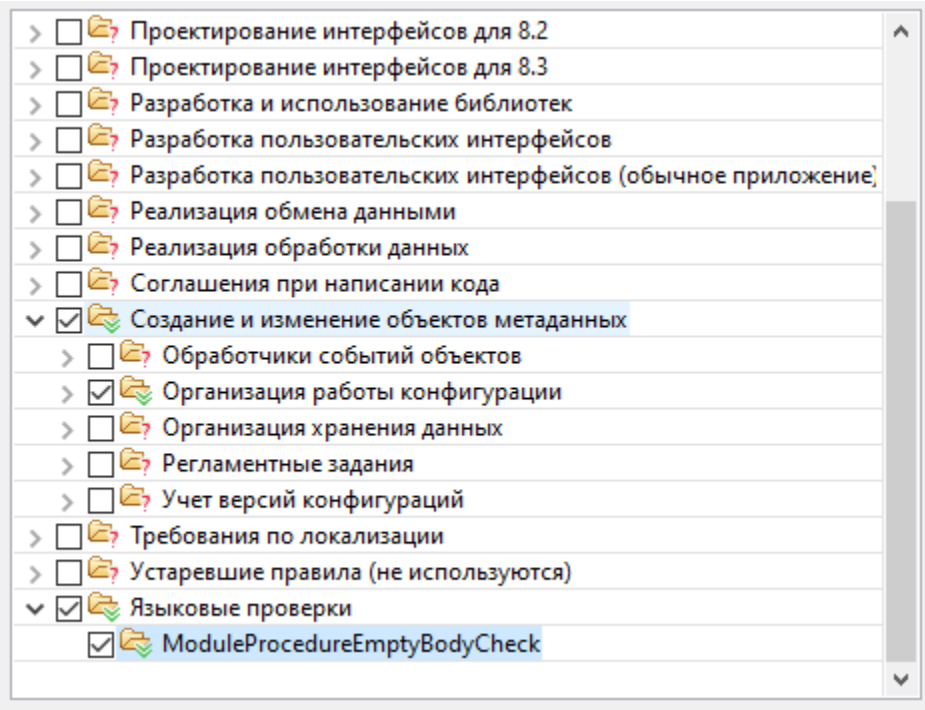
Система автоматически контролирует граничные условия, такие как:

- Автоматическое создание профиля по умолчанию в случае отсутствия профилей (или ошибки их загрузки)
- Наличие несохраненных изменений в текущем профиле и предотвращение переключения профиля в этом случае

Все вносимые изменения настроек проверок применяются к текущему (выбранному) профилю настроек. Поскольку операция проверки - достаточно дорогостоящая, то 1С:EDT применяет внесенные изменения настроек только при нажатии кнопки "Применить" (или "Применить и закрыть"). Вместе с тем, 1С:EDT поддерживает селективный перезапуск проверок, минимизируя область проверок только минимально необходимым набором проверок, чьи настройки были изменены.

1.2.4.2. Дерево проверок

Все проверки, зарегистрированные в системе (как встроенные в 1С:EDT, так и установленные сторонние) доступны в виде дерева в окне настроек проверок:



Для удобства работы с большим количеством проверок, они разбиты по соответствующим многоуровневым категориям.

1.2.4.3. Поиск проверки

Для поиска необходимой проверки, пользователь может воспользоваться фильтром проверок, расположенным над списком проверок в окне настроек.

Кроме того, существует быстрый вариант перехода от маркера к создавшей его проверки. Для этого пользователю необходимо нажать ПКМ на соответствующей записи в окне "Ошибки конфигурации", и выбрать пункт выпадающего меню "Перейти к проверке".

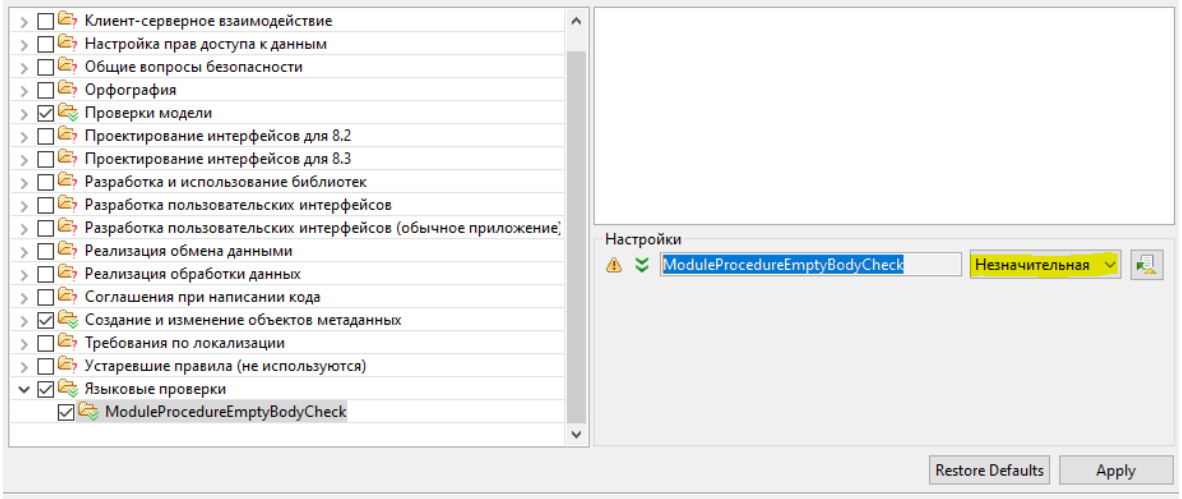
Это действие автоматически откроет страницу настроек с фокусом на нужно проверке.

1.2.4.4. Редактирование настроек проверки

Проверки в 1С:EDT позволяют пользователю настраивать их поведение, а именно:

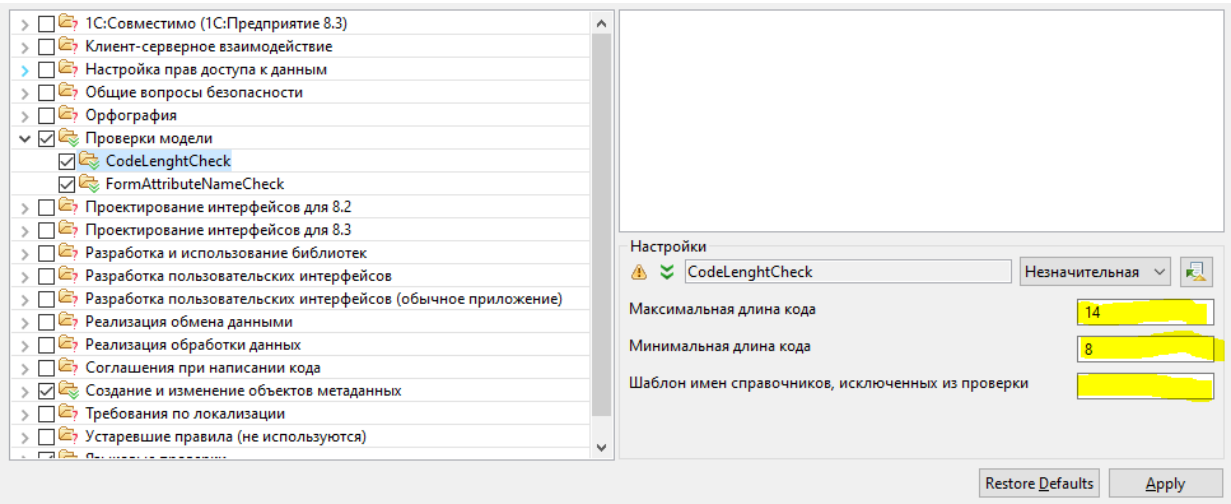
- Менять критичность маркеров для обнаруженных проверкой проблем
- Изменять значения специфичных параметров проверки
- И возвращать настройки к значениям по умолчанию

Для смены **критичности маркеров** проблем служит следующий комбо-бокс:



При смене критичности маркеров и последующего подтверждения изменения настроек проверки, система автоматически обновляет все ранее созданные маркеры без перезапуска процесса валидации (т.е. оперативно).

Уникальные параметры проверки представлены в виде набора строковых полей:



Пользователь может вводить любые валидные значения в параметры. В случае, если введены невалидные данные, параметр игнорируется. Вместо этого используется значение по умолчанию, заданной в самой проверке.

1.2.4.5. Отключение/Включение проверки

Также пользователь может решить:

- Полностью отключить проверку в данном профиле
- Либо включить ранее отключенную проверку (самим пользователем, или отключенную разработчиком по умолчанию)

Для этого пользователю необходимо выбрать соответствующий элемент в дереве проверок и нажать на чекбокс, находящийся рядом с иконкой проверки.

Система автоматически перезапустит необходимые проверки после подтверждения внесения изменений.

1.2.5. Подавление результатов проверок

Иногда встречается необходимость не просто отключить проверку целиком. Вместо этого необходимо убрать одиночные маркеры, создаваемые проверкой в контексте определенных объектов (например). Очень часто такие требования предъявляются к различным библиотечным решениям, где некоторые отступления от стандартов кодирования сделаны осознанно, с целью обеспечения правильной компонентизации решения.

Для решения подобных задач служит механизм подавления результатов проверок.

Поскольку природа данных конфигурации двойственна (объекты и языковые модули), соответственно проверки, а значит и их подавления, также имеют определенную специфику.

Поэтому в рамках системы подавлений проверок есть два практически независимых компонента:

- Механизм подавления результатов объектных проверок
- И механизм подавления результатов языковых проверок

С особенностями двух данных компонент мы и познакомимся далее.

1.2.5.1. Механизм подавления результатов объектных проверок

Ключевые возможности системы настроек подавления ошибок для механизма валидации:

- Хранение настроек в дереве ресурсов соответствующих проектов 1C:EDT
- Полная интеграция в механизм валидации

См. полное описание с примерами настроек для объектов разного типа в соответствующем приложении.

1.2.5.2. Пользовательский интерфейс подавлений объектных проверок

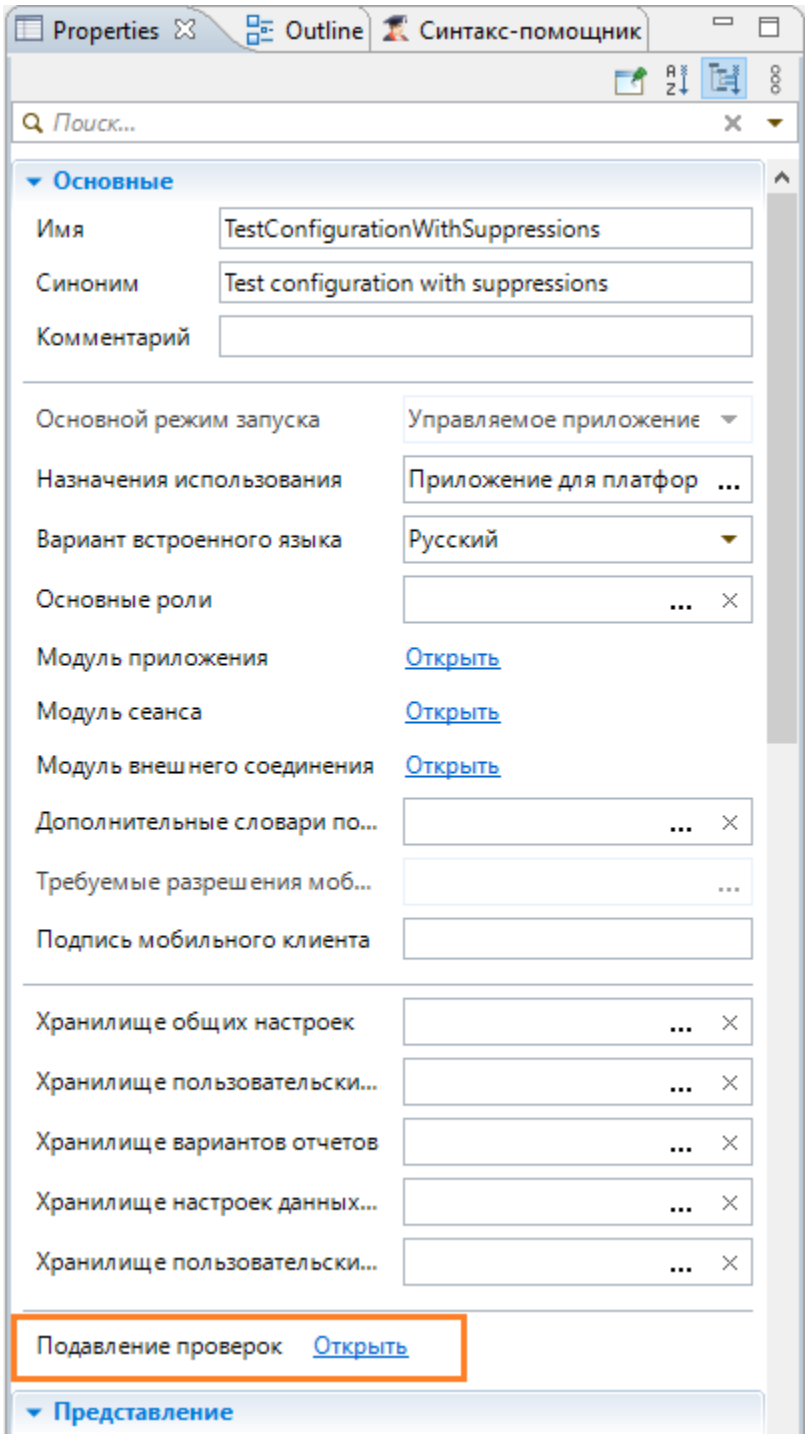
Для удобства настроек объектных подавлений был реализован упрощенный пользовательский интерфейс (в дальнейшем **UI**). Ограничения в основном касаются поддерживаемой в модели подавлений концепции наследования и перезаписи настроек, которая позволяет как устанавливать подавления на разных уровнях дерева объектов, так и отменять признаки подавления у выставленных выше настроек. В UI есть возможность **только подавлять** проверки, но не отменять подавления.

В UI не поддерживаются настройки подавления модулей. Для модулей реализован другой подход, основанный на комментариях к семантическим элементам языковых модулей. См. ниже.

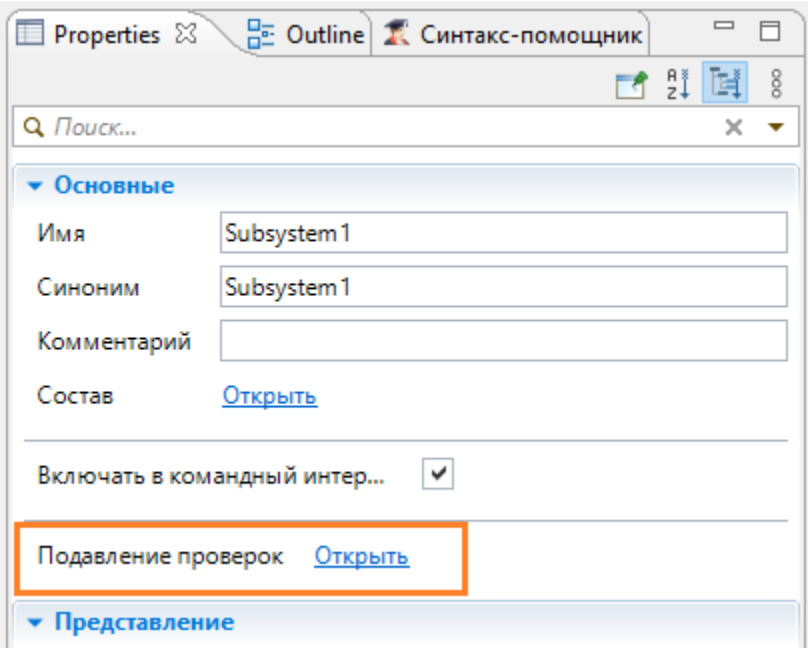
1.2.5.2.1. Создание/модификация настроек подавления на объекте

Для доступа к редактору подавлений необходимо выбрать интересующий объект в навигаторе, а затем в палитре свойств кликнуть по лике "Открыть" у свойства **"Подавление проверок".**

У конфигурации:



У подсистемы:

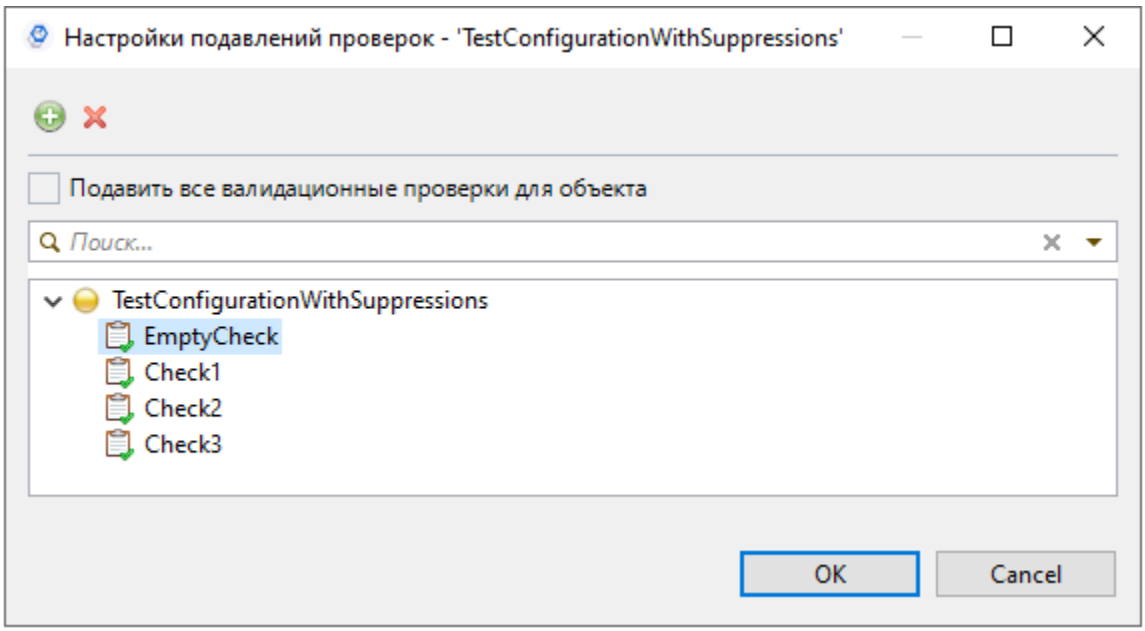


1.2.5.2.2. Редактор настроек подавлений

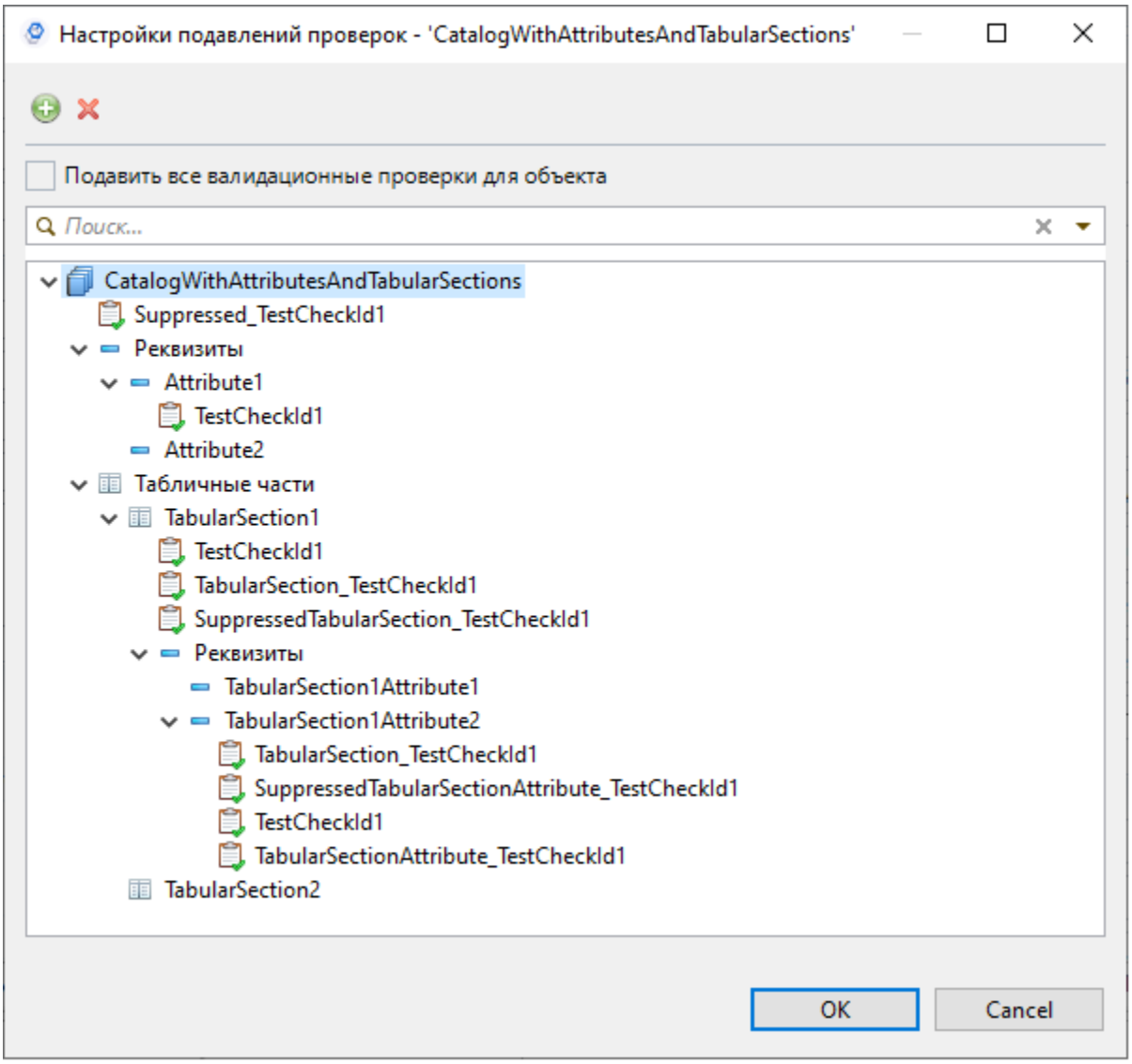
Редактор представляет собой дерево метаданных с поиском и фильтрацией, двумя кнопками (добавить и удалить подавления) и чек-бокс для подавления всех проверок на уровне объекта.

Редактор подавлений на уровне конфигурации. Здесь можно добавлять и удалять подавления для объекта-конфигурации. Удалить можно как отдельную проверку, так и все проверки объекта. Для удаления всех проверок нужно встать в дереве на нод объекта и нажать кнопку *Удалить*.

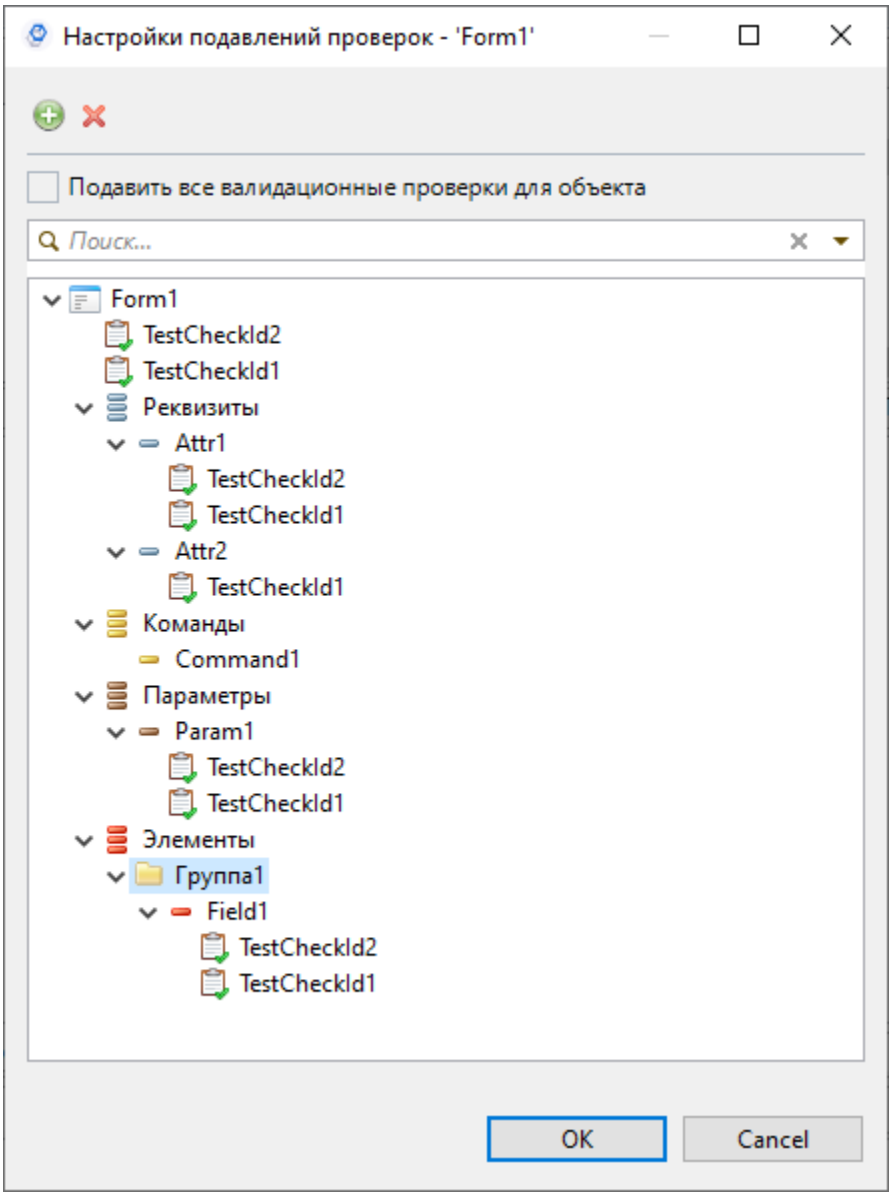
Для редакторов с вложенными объектами (containments) это правило тоже работает - удаляются все проверки, лежащие в выбранном узле объекта и во всех вложенных объектах.



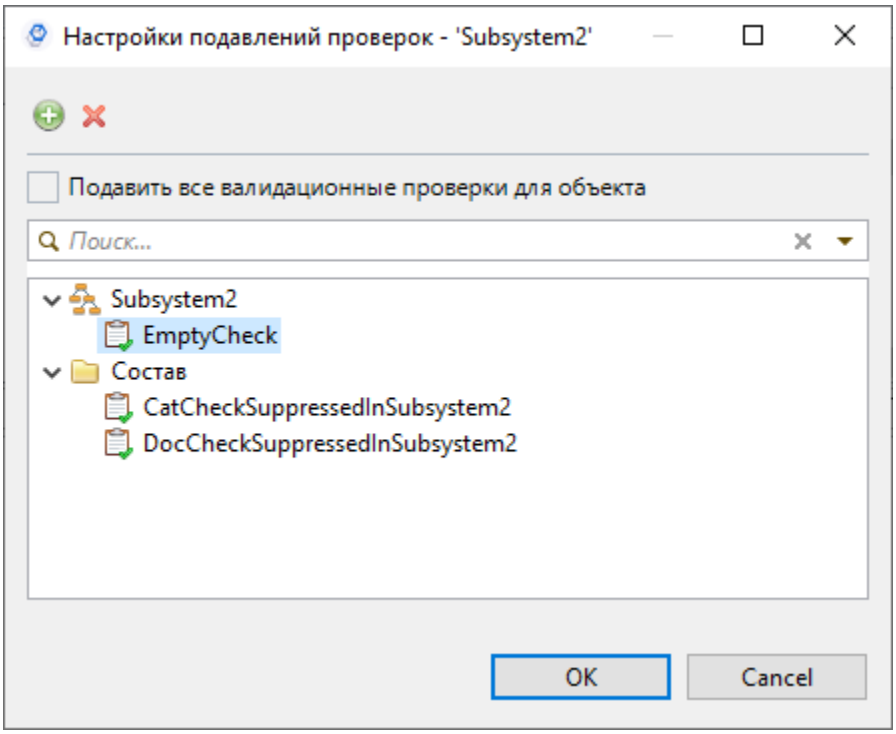
Редактор подавлений на уровне справочника. Здесь можно манипулировать составом проверок не только на самом справочнике, но и на его реквизитах, табличных частях и реквизитах табличных частей.



Редактор подавлений на уровне формы. Здесь можно управлять составом проверок на уровне всей формы, ее реквизитов, команд, параметров и элементов.



Редактор подавлений на уровне подсистемы. Здесь, помимо настроек подавлений на уровне самого объекта-подсистемы, можно еще настраивать подавления для ее состава.



1.2.5.3. Механизм подавления результатов языковых проверок

Механизм подавления результатов языковых проверок основан на комментариях к семантическим элементам, добавляемых непосредственно в исходный код языкового модуля.

1.2.5.3.1. Формат описания подавлений в языковых модулях

Синтаксис комментария подавления проверок:

//@skip-ckeck <Код-1>, <Код-2>, . . . [- необязательный комментарий с описанием причины подавления]

- Подавление на уровне модуля

```
<//   1 |  >
...
<//   N |  >
// @skip-check <-1>, <-2>, <-3> - <>
<//   N+2 |  >

<  >

:

//  , ,

// @skip-check NStr-contains-camelcase -            CamelCase

//
 ()
...
 = ("ru = '    ';");
...
```

- Подавление на уровне метода

```
< >

// @skip-check <-1>, <-2>, <-3> - <>

...


:

//
// @skip-check NStr-contains-camelcase -             CamelCase
 ()
...
 = ("ru = '    ';");
...
```

- Подавление на уровне statement'a

```
// C       ,

//  -  statement-,   ()
// @skip-check <-1>, <-2>,<-3> - <>
<statement>

//  - inline
<statement> // @skip-check <-1>, <-2>,<-3> - <>

//     ACC:1036
// @skip-check ACC:1036 -
(, );

//     - ACC:1036  ACC:1037
//@skip-check ACC:1036, ACC:1037 -
(, );

//     ACC:1036, inline
(, ); // @skip-check ACC:1036 -

//     - ACC:1036  ACC:1037, inline
(, ); //@skip-check ACC:1036, ACC:1037 -
```

- Подавление на уровне блока кода

```
//  ,
// @skip-check <-1>, <-2>, <-3> - <>
 ()

//  , ,   .
// @skip-check <-1>, <-2>, <-3> - <>
#
...
#

:

// @skip-ckeck Nstr-contains-camaelcase -
 ()
...
 = ("ru = '    ';");
-------------------------------------


// @skip-ckeck Nstr-contains-camaelcase -
#

 ()
...
 = ("ru = '    ';");
...
 2()
...
 = ("ru = '     2';");
...
#
-------------------------------------


//
 ()
...

// @skip-ckeck Nstr-contains-camaelcase -
#

...
 = ("ru = '    ';");
...
#
...
```

1.2.6. Отчет о результатах проверки

2. Проверки объектов модели метаданных

В предыдущих разделах мы ознакомились с результатами работы подсистемы проверок 1C:EDT, ее настройками и возможностями по управлению.

Теперь же перейдем непосредственно к обзору возможностей подсистемы проверок 1C:EDT для разработки и поддержки пользовательских/сторонних проверок

2.1. Пример 1 - проверка длины кода справочника

В качестве первого примера для демонстрации возможностей по разработке проверок выберем проверку длины кода справочника со следующими возможностями:

- Настраиваемые границы допустимого значения длины кода (верхняя и нижняя)
- Фильтр по имени справочника для исключения части справочников из проверок

2.1.1. Подготавливаем среду разработки

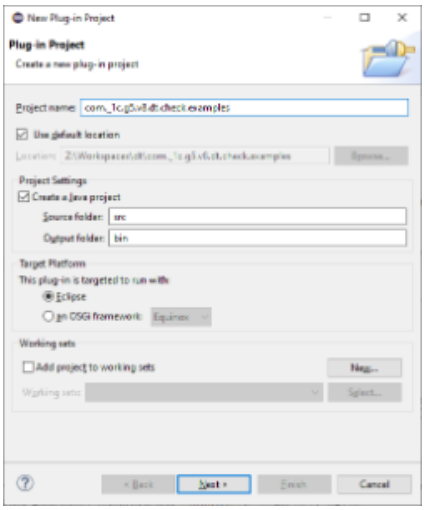
Для разработки расширений 1С:EDT (в том числе - содержащих проверки) необходимо использовать среду разработки расширений 1С:EDT (ссылка). Особенности установки и настройки описаны в соответствующей документации.

Не пытайтесь разрабатывать проверки в самой 1С:EDT - это среда разработки приложений платформы 1С, а не среда разработки расширений самой 1С:EDT

2.1.2. Создаем новый бандл для проверок

Проверка поставляется и интегрируется в 1С:EDT в виде стандартного Eclipse-бандла. Для целей обучения создадим бандл **com._1c.g5.v8.dt.check.examples**.

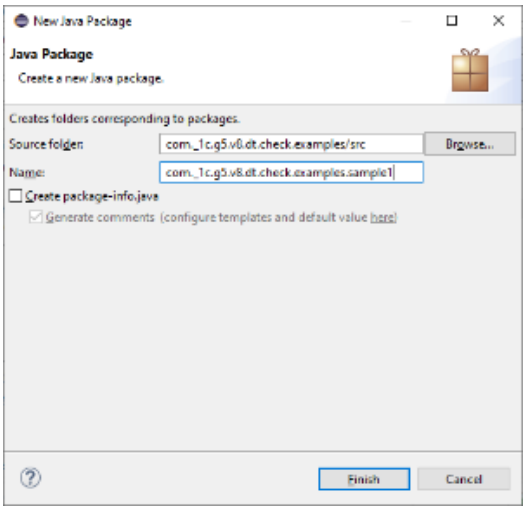
Для этого воспользуемся мастером создания плагинов Eclipse (**New/Plug-in Project**):



В результате мы получаем новый проект **com._1c.g5.v8.dt.check.examples,** уже открытый в нашем рабочем пространстве

2.1.3. Выбираем имя пакета для проверки

Разрабатываемые в рамках данного гайда проверки структурированы в соответствии с соответствующим примером. Поэтому для первой проверки создадим пакет **com._1c.g5.v8.dt.check.examples.sample1**:



2.1.4. Выбираем базовый класс/интерфейс проверки

Подсистема проверок 1С:EDT обеспечивает разработку и поддержку как весьма сложных и неоднозначных сценариев, так и простых/массовых проверок. Поэтому для разработчиков предоставляется несколько уровней API:

- Основной API разработки проверок, с доступом ко всем возможностям подсистемы
- И набор стандартных базовых проверок, предоставляющих возможность быстрой реализации проверок для множества стандартных сценариев

Разумеется, первый вариант отличается повышенной сложностью при разработке, поскольку разработчику придется учитывать все аспекты работы проверки. Поэтому разработку каждой новой проверки рекомендуется начинать с проверки набора стандартных проверок и их возможностей, как значительно экономящих время при разработке в случае, когда возможностей базовой проверки хватает для реализации поставленной задачи.

Базовые проверки находятся в package-е **com._1c.g5.v8.dt.check.components:**

- **BasicCheck**

BasicCheck удовлетворяет требованиям нашей задачи, поэтому наша проверка будет базироваться на базе этой реализации.

2.1.5. Добавляем зависимости в новый бандл

Разработку мы начнем с установки необходимого минимума зависимостей для нашего нового бандла с проверками. Откроем файл MANIFEST.MF и добавим следующие зависимости:

```
Require-Bundle: org.eclipse.core.runtime;bundle-version="[3.11.1,4.0.0)",
 org.eclipse.emf.ecore;bundle-version="[2.12.0,3.0.0)"
Import-Package: com._1c.g5.v8.dt.check.components;version="[1.0.0,2.0.0)",
 com._1c.g5.v8.dt.check;version="[1.0.0,2.0.0)",
 com._1c.g5.v8.dt.metadata.mdclass;version="[8.0.0,9.0.0)"
```

Где:

- com._1c.g5.v8.dt.check - основной публичный API подсистемы проверок
- com._1c.g5.v8.dt.check.components - набор стандартных реализаций и хелперов для быстрой разработки типовых проверок
- com._1c.g5.v8.dt.metadata.mdclass - бандл модели метаданных конфигурации (поскольку мы собираемся разрабатывать проверку именно для метаданных конфигурации)

На этом приготовления закончены, и мы можем переходить непосредственно к разработке.

2.1.6. Знакомимся со структурой BasicCheck

Создадим нашу проверку (назовем ее **CatalogCodeLenghtCheck**) и унаследуем ее от **BasicCheck**:

```
/**
 * Checks the catalog's code length
 */
public class CatalogCodeLenghtCheck
    extends BasicCheck
{
    @Override
    public String getCheckId()
    {
        return null;
    }

    @Override
    protected void configureCheck(CheckConfigurer configurationBuilder)
    {
    }

    @Override
    protected void check(EObject object, ResultAcceptor resultAceptor, ICheckParameters parameters,
        IProgressMonitor progressMonitor)
    {
    }
}
```

Как мы можем видеть, минимальная проверка на базе **BasicCheck** должна реализовать три метода для того, чтобы считаться рабочей. Познакомимся с ними подробнее:

- **getCheckId** - возвращает уникальный **код проверки**, используемый системой проверок для поиска и адресации результатов проверки. Данный код задается на этапе разработки, и не меняется настройками пользователя для проверки
- **configureCheck** - конфигурирует проверку в рамках 1C:EDT для:
  - Правильной реакции на события об изменении данных конфигурации (и последующего автоматического запуска проверки)
  - Предоставления 1C:EDT информации о параметрах проверки
  - Описательная информация для отображения на интерфейсе управления проверкам 1C:EDT
- **check** - собственно логика проверки. Получает на вход целевой объект (определяемый правилами реакции проверки на изменения, задаваемые в **configurationCheck**), обеспечивая проверку и запись результатов (если таковые имеются)

Начнем реализовывать каждый метод, останавливаясь подробнее на деталях и требованиях каждой реализации.

2.1.7. Выбираем и устанавливаем код проверки

При выборе кода проверки, необходимо руководствоваться стандартами 1C:EDT по формированию кодов проверки (*документ в разработке*).

Для нашей тестовой проверки выберем код "**CatalogCodeLenghtCheck**", и зададим его в коде проверки:

```
public class CatalogCodeLenghtCheck
    extends BasicCheck
{
    /**
     * This check unique indentifier
     */
    public static final String CHECK_ID = "CatalogCodeLenghtCheck"; //$NON-NLS-1$

    @Override
    public String getCheckId()
    {
        return CHECK_ID;
    }
...
```

Данный код будет зарегистрирован в 1C:EDT при загрузке нашего бандла (об этом - далее), и именно этот код будет использоваться для локации результатов проверки.

2.1.8. Конфигурируем проверку для автоматического запуска

Для обеспечения автоматического определения области действия проверки (объект, свойство и т.д.) и последующего автоматического запуска в случае изменения данных (либо первого их появления), необходимо сконфигурировать проверку, пользуясь возможностями конфигурационного объекта **CheckConfigurer**, передаваемого в нашу проверку подсистемой проверок 1C:EDT в момент регистрации новой проверки.

Для описания правил отслеживания изменений модели данных, **BasicCheck** использует концепцию иерархического описания структуры модели, представленного в коде в виде объектного билдера. Зададим правило отслеживания изменений свойства "Длина кода" объекта метаданных типа "Справочник" с помощью данного механизма, и разберем пример в деталях:

```
...
import static com._1c.g5.v8.dt.metadata.mdclass.MdClassPackage.Literals.CATALOG;
import static com._1c.g5.v8.dt.metadata.mdclass.MdClassPackage.Literals.CATALOG__CODE_LENGTH;
...
public class CatalogCodeLenghtCheck
    extends BasicCheck
{
    @Override
    protected void configureCheck(CheckConfigurer configurer)
    {
        //@formatter:off
        configurer.
            topObject(CATALOG). //
                features(CATALOG__CODE_LENGTH); //     :
                                                // -
                                                // -   ,
                                                // -      " "
        //@formatter:on
    }
...
```

Как мы видим, построитель конфигурации проверки оперируют терминами "объект верхнего уровня" (topObject), свойство (feature) и т.п., которые не являются интуитивно понятными. И теперь самое время немного отклониться в сторону, и познакомиться с некоторой спецификой фреймверка EMF, используемого в качестве базиса для механизма хранения и доступа к персистентным моделям 1C:EDT (так называемый BM/БМ - BigModel/БольшаяМодель).

2.1.9. Базовые понятия EMF и БМ, используемые при конфигурировании проверок

Для эффективной разработки проверок, необходим как минимум базовый уровень понимания терминов и основных элементов технологии хранения и доступа к данным 1C:EDT.

Все модели данных конфигурации в 1C:EDT (метаданные, формы, языковые модули) представлены в системе в виде EMF-моделей (Eclipse Modeling Project | The Eclipse Foundation). EMF - это framework для быстрого проектирования и генерации объектных моделей по некоторым метаописаниям. Основные моменты решения, важные для разработки проверок:

- Каждый EMF объект представлен вполне конкретным Java-классом, генерируемым системой генерации кода EMF, и допускают прямые операции с ними (т.е. чтение и запись свойств, навигация по ссылкам и т.п.)
- При этом EMF-модели параллельно с явным доступом к данным, обеспечивают рефлективный доступ по соответствующим метаописаниям классов и свойств - EClass-ам и feature-ам соответственно
- Данные объекты в зависимости от природы источника (например метаданные и языковые данные) могут хранится в различных объектных хранилищах/производится на лету, что накладывает определенные ограничения на взаимодействия с ними (об этом - подробнее в примерах)

1C:EDT использует две основных разновидности хранилищ EMF-данных:

- Это основное персистентное объектное хранилище данных, так называемая БМ (ссылка на описание). В ней хранятся практически все данные, кроме языковых модулей и запросов
- И языковые данные, с динамической загрузкой из исходных модулей конфигурации

При работе с основным объектным хранилищем существуют следующие особенности:

- **Основные объекты** конфигурации (метаданные, макеты, формы и т.п.) представлены в виде объектов верхнего уровня (top objects), содержащих поддеревья **подчиненных** (containment) объектов и примитивных листовых свойств. Объекты верхнего уровня адресуемы по их полностью квалифицированным именам. либо могут быть получены из соответствующих ссылок в соответствующих местах модели
- Работа с данными объектами *всегда* должна вестись в транзакции БМ. За исключением определенных случаев (см. примеры), подсистема проверок автоматически заботится об этом, подключая проверки к соответствующей транзакции проверки
- Полученные из других источников объекты (например из языковых модулей) объекты модели метаданных/внешних свойств должны быть подключены к транзакции прежде, чем с ними будут произведены какие-то операции
- Хранилище отслеживает все изменения, происходящие с объектами, генерируя соответствующие события для всех подписчиков. Собственно, это основной механизм, на котором заждется механизм автоматического запуска проверок. И именно эти правила (реакции на те или иные изменения модели) и описываются при разработке проверки (в том, или ином виде)

При работе с языковыми данными существуют следующие особенности:

- Модели модулей загружается из живого исходника (предварительно осуществлен расчет всех важных производных данных)

- Переходы на объекты метаданных работают (например - получение владельца модуля из модели модуля), но полученные объекты модели метаданных/внешних свойств не привязаны к конкретной транзакции. Для работы с ними необходимо произвести привязку (об этом - в примерах)
- Процесс загрузки модулей достаточно дорогой, поэтому при реализации проверок крайне нежелательно делать кросс-модульные проверки с ручной загрузкой дополнительных модулей и т.п. Вместо этого предлагается использовать всю доступную производную информацию (система типов и т.п.) для обеспечения независимой проверки отдельных модулей, но с учетом общего состояния данных конфигурации

2.1.10. Конфигурация проверки CatalogCodeLenghtCheck - продолжение

Ознакомившись с основными понятиями EMF и БМ, теперь мы можем осознать, какие именно правила заданы в нашей конфигурации проверки:

- topObject(CATALOG) - 1C:EDT будет проверять все изменения БМ, реагируя только на изменения данных в объекте верхнего уровня, имеющего **EClass MdClassPackage.Literals.CATALOG.** Соответствующий объект модели - это **com._1c.g5.v8.dt.metadata.mdclass.Catalog,** т.е. объект модели 1C:EDT, представляющий объект метаданных **Справочник** в платформе
- features(CATALOG__CODE_LENGTH) - список свойств (feature) объекта с **EClass MdClassPackage.Literals.CATALOG**, на изменение которых будет запущена проверка. В нашем случае - это одно свойство (**MdClassPackage.Literals.CATALOG__CODE_LENGTH**), соответствующее свойству "Длина кода" объекта метаданных **Справочник.** В данной упрощенной реализации подразумевается, что запуск проверки будет осуществляться при *любом* изменении данного свойства, а именно:
    - Первоначальная инициализация поля при создании нового справочника (как ручное создание, так и импорт из ИБ и т.п.)
    - Любое изменение данного поля

На этом конфигурирование реактивной части проверки завершено, и мы можем переходить непосредственно к методу проверки, где, собственно, и будет видно, как предоставленные настройки повлияют на саму проверку.

2.1.11. Реализуем логику проверки CatalogCodeLenghtCheck

Собственно логика данной проверки достаточно прямолинейна, хотя и не лишена определенных особенностей:

```
...
    private static final int MIN_LENGTH = 8;
    private static final int MAX_LENGTH = 14;

    /*
     *      .
     */
    @Override
    protected void check(EObject object, ResultAcceptor resultAcceptor, ICheckParameters parameters,
        IProgressMonitor progressMonitor)
    {
        Catalog catalog = (Catalog)object;

        int numberLength = catalog.getCodeLength();

        if (numberLength < MIN_LENGTH || numberLength > MAX_LENGTH)
        {
            String errorMessage = numberLength > MAX_LENGTH ? "    " //$NON-NLS-1$
                : "    "; //$NON-NLS-1$
            resultAcceptor.addIssue(errorMessage, CATALOG__CODE_LENGTH);
        }
    }

...
```

Остановимся на наиболее важных моментах:

- Catalog catalog = (Catalog)object; - поскольку мы явно задали правило, по которому наша проверка может быть вызвана только на объекте типа Справочник, и только при изменении его свойства CodeLength - мы спокойно можем привести тип входящего на проверку объекта к Catalog - никакие другие типы объектов сюда попасть не могут (за этим следит сама 1C:EDT)
- int numberLength = catalog.getCodeLength(); - получение текущего (обновленного или начального) значения длины кода справочника. Здесь интересными являются следующие моменты:
    - Несмотря на все то, что мы писали ранее о рефлективной модели EMF, при обычной работе с собственно моделями (а не их метаописанием) нет никакой надобности использовать рефлективную природу EMF, можно напрямую работать с полученным объектом, его свойствами и методами
    - Приходящий в проверку объект автоматически подключается к read-only транзакции БМ, гарантируя актуальность данных (раз), и защиту от непреднамеренных изменений в процессе проверки (два)

- Для регистрации проблемной ситуации служит объект **ResultAcceptor** , принимающий на вход различные варианты проблем. В данном случае регистрируемая проблема привязывается к полю справочника опять за счет той же самой feature, которую мы уже рассматривали ранее. В результате система знает, в каком именно поле была найдена проблема, чем обеспечивает механизм отображения маркеров проблем информацией для адресного перехода (объект и его свойство)

На этом, собственно, основная логика самой проверки завершена, осталось зарегистрировать ее в реестре проверок и проверить работоспособность.

2.1.12. Регистрируем новую проверку в 1C:EDT через точку расширения

Наша первая проверка готова, и теперь нам необходимо сообщить подсистеме проверок 1C:EDT о ее существовании. Для этого используется точка расширения **com._1c.g5.v8.dt.check.checks**.

Для регистрации первой проверки нам нужно указать как минимум два элемента:

- Категорию, в которую входит проверка. Данная категория будет участвовать в формировании дерева категорий и навигации по всем зарегистрированным проверкам
- И, собственно указать саму проверку

Точка расширения задается в файле **plugin.xml.** Создадим его и введем в него данные:

```
<plugin>
         <extension point="com._1c.g5.v8.dt.check.checks">
      <category
          id="com._1c.g5.v8.dt.check.checks.ModelCategory"
          title="Model Checks"
          category="">
      </category>

      <check
          category="com._1c.g5.v8.dt.check.checks.ModelCategory"
          class="com._1c.g5.v8.dt.check.examples.example1.CatalogCodeLenghtCheck">
      </check>
   </extension>
</plugin>
```

Остановимся на основных моментах регистрации проверок:

- Описание категорий проверок
  - Категория задается тегом **<category>**
  - Идентификатор категории **не должен** быть человекочитабельным. Вместо этого он должен следовать требованиям по уникальности. Сам идентификатор категории не отображается в UI и не используется для каких-либо пользовательских операций
  - Заголовок категории задается атрибутом **title**. Могут использоваться стандартные локализации Eclipse (см. приложение)
- Регистрация проверки
  - Проверка регистрируется тегом **<check>**
  - Обязательно указание категории, посредством установки ее идентификатора в атрибут **category**. В случае, если категория не указана - проверка не будет доступна
  - Класс проверки указывается через атрибут **class**. Для проверок, использующих в своей работе сервисы 1C:EDT, необходимо воспользоваться фабриками расширений (*документ в разработке*)

Примечание - не забудьте включить созданный plugin.xml в соответствующий build.properties вашего бандла. иначе при сборке бандла и инсталляции вы не увидите вашей проверки:

```
source.. = src/
output.. = bin/
bin.includes = META-INF/,\
               .,\
               plugin.xml
```

2.1.13. Тестируем проверку

Для тестирования проверки 1C:EDT предоставляет базовый инструментарий интеграционного тестирования. Он располагается в бандле **com._1c.g5.v8.dt.testing.check**

Для создания нового теста, необходимо:

- Создать ваш тестовый бандл
- Подготовить тестовую конфигурацию, и поместить ее в каталог workspaces в корне вашего тестового бандла
- Создать новый тестовый класс, унаследованный от com._1c.g5.v8.dt.testing.check.CheckTestBase
- Используя CheckTemplateTest.java в качестве пособия, реализовать ваши собственные тестовые сценарии

2.2. Пример 2 - параметризованная проверка длины кода справочника

В предыдущем разделе мы реализовали простой пример проверки длины кода. Однако у данной проверки есть один серьезный недостаток - данная проверка жестко привязана к специфике отдельных конфигураций и не позволяет настраивать проверку для использования в других конфигурациях (например, с другими требованиями к длине кода). Исправим эту ситуацию, и обновим нашу проверку, добавив в нее параметры.

2.2.1. Добавляем параметры в описание проверки

Параметры добавляются через тот же объект конфигурирования, что и другие элементы конфигурации проверки:

```
...
    private static final String MIN_LENGTH_PARAMETER_NAME = "minLength"; //$NON-NLS-1$
    private static final String MAX_LENGTH_PARAMETER_NAME = "maxLength"; //$NON-NLS-1$
    private static final String EXCLUDE_NAME_PATTERN_PARAMETER_NAME = "excludeNamePattern"; //$NON-NLS-1$
...
public class CatalogCodeLenghtCheck
    extends BasicCheck
{
    @Override
    protected void configureCheck(CheckConfigurer configurer)
    {
        //@formatter:off
        configurer.
            topObject(CATALOG). //
                features(CATALOG__CODE_LENGTH). //     :
                                                // -
                                                // -   ,
                                                // -      " "
            parameter(MIN_LENGTH_PARAMETER_NAME, Integer.class, "8", Messages.
CatalogCodeLengthCheck_MinLengthParameter_Title). //$NON-NLS-1$
            parameter(MAX_LENGTH_PARAMETER_NAME, Integer.class, "14", Messages.
CatalogCodeLengthCheck_MaxLengthParameter_Title). //$NON-NLS-1$
            parameter(EXCLUDE_NAME_PATTERN_PARAMETER_NAME, String.class, "", Messages.
CatalogCodeLengthCheck_ExcludePattern_Title); //$NON-NLS-1$
        //@formatter:on
    }
...
```

Зададим следующие параметры:

- Минимальная длина кода проверки, со значением по умолчанию, равным 8
- Максимальную длину кода проверки, со значением по умолчанию, равным 14
- И регулярное выражение, служащее для исключения определенных справочников из проверки по их имени, с пустым значением по умолчанию

Собственно этого уже достаточно для того, чтобы мы увидели данные параметры на интерфейсе управления настройками проверок.

2.2.2. Использование параметров

Для получения значений параметров в проверке используется контекстный объект с интерфейсом ICheckParameters, передаваемый в проверку во время ее выполнения:

```
...
    private static final String MIN_LENGTH_PARAMETER_NAME = "minLength"; //$NON-NLS-1$
    private static final String MAX_LENGTH_PARAMETER_NAME = "maxLength"; //$NON-NLS-1$
    private static final String EXCLUDE_NAME_PATTERN_PARAMETER_NAME = "excludeNamePattern"; //$NON-NLS-1$
...
public class CatalogCodeLenghtCheck
    extends BasicCheck
{
    /*
```

```
     *      .
     */
    @Override
    protected void check(EObject object, ResultAcceptor resultAcceptor, ICheckParameters parameters,
        IProgressMonitor progressMonitor)
    {
        Catalog catalog = (Catalog)object;

        //        ,
        // ,       /WS
        int minLength = parameters.getInt(MIN_LENGTH_PARAMETER_NAME);
        int maxLength = parameters.getInt(MAX_LENGTH_PARAMETER_NAME);
        String excludeNamePattern = parameters.getString(EXCLUDE_NAME_PATTERN_PARAMETER_NAME);

        //  ,        -
        //   -
        if (excludeNamePattern != null && catalog.getName().matches(excludeNamePattern))
        {
            return;
        }

        ...
    }
...
```

Для получения значений параметров необходимо:

- Обратиться к объекту **parameters** по методу, соответствующему ранее установленному в конфигурации проверки типу параметра
- Использовать полученное значение в логике проверки

2.3. Пример 3 - проверка имени реквизитов справочника и реквизитов табличных частей справочника

К настоящему моменту, мы уже ознакомились с тем, каким образом можно провалидировать свойства объектов верхнего уровня. Однако, что же делать, если мы захотим проверить свойства объектов, вложенных в объекты верхнего уровня (например реквизиты. табличные части и их реквизиты)?

На этот вопрос ответит наш следующий пример - проверка имени реквизитов справочника и реквизитов табличных частей справочника. В данной проверке появляются новые для нас элементы:

- Проверка свойств вложенных объектов
- Проверка более чем одного свойства, при этом - свойств разнотипных объектов

Определим логику нашей проверки:

- Область действия
    - Тип объекта верхнего уровня - справочник
- Проверяемые свойства:
    - Имя реквизита справочника
    - Имя реквизита табличной части справочника
- Логика проверки
    - Имя элемента не должно начинаться с символа '_'
- Параметризация
    - Паттерн для имени справочников, элементы которых должны быть исключены из данной проверки

2.3.1. Создаем новую проверку

Документ в разработке

2.3.2. Валидация свойств вложенных объектов

Ранее мы задавали правила для запуска проверки только для свойств объектов верхнего уровня. Теперь зададим правила запуска проверки для вложенных объектов:

```
public class CatalogAttributeNameCheck
    extends SingleObjectModelCheck<CatalogAttribute>
```

```
{
    //  ,
    private static final String EXCLUDE_NAME_PATTERN_PARAMETER_NAME = "excludeNamePattern"; //$NON-NLS-1$
   ...
    @Override
    protected void configureCheck(CheckConfigurer builder)
    {
        //@formatter:off
        builder.topObject(CATALOG).
            containment(CATALOG_ATTRIBUTE).
                features(MD_OBJECT__NAME).
            containment(TABULAR_SECTION_ATTRIBUTE).
                features(MD_OBJECT__NAME).
            parameter(EXCLUDE_NAME_PATTERN_PARAMETER_NAME, String.class, "", Messages.
CatalogAttributeNameCheck_ExcludePattern_Title); //$NON-NLS-1$
        //@formatter:on
    }
  ...
```

Разберем данное описание правил детально:

- topObject(CATALOG), как и ранее, указывает, что проверка будет выполняться для всех объектов конфигурации, имеющих **EClass** CATALOG
- containment(CATALOG_ATTRIBUTE) - а вот это уже новый для нас элемент. Он указывает на тип объекта, содержащегося в соответствующем объекте верхнего уровня. Данная конструкция означает, что в проверке будут участвовать все объекты данного типа (в нашем случае - это реквизиты справочника), а также их наследники (если таковые есть). Это дает некоторые дополнительные возможности по организации проверок по иерархии объектов (например проверок имен MD_OBJECT-ов и т.п.)
- features(MD_OBJECT__NAME) - подчиненный ранее объявленному containment()-у элемент, который задает (как и в предыдущих примерах) правило отслеживания изменений в свойстве реквизита справочника "имя"
- Второй элемент containment(TABULAR_SECTION_ATTRIBUTE) - ожидаемо задает правило выполнения проверки для реквизитов табличных частей. Система проверок 1C:EDT сама заботится о поиске соответствующих объектов в объекте верхнего уровня
- И уже для этого правила мы опять же указываем конкретное свойство, изменение которого мы отслеживаем (как и выше - это имя объекта метаданных)

В данном случае правила запуска проверки уже заметно сложнее, чем ранее, и требуют понимания тех основных понятий EMF, о которых мы рассказывали ранее.

2.3.3. Разнотипные объекты, приходящие в проверку

Так как наша проверка теперь поддерживает проверку не одного, а двух типов объектов, нам необходимо учесть это в реализации самой проверки:

```
public class CatalogAttributeNameCheck
    extends BasicCheck
{
   ...
    @Override
    protected void check(EObject object, ResultAcceptor resultBuilder, ICheckParameters parameters,
        IProgressMonitor progressMonitor)
    {
        String excludeNamePattern = parameters.getString(EXCLUDE_NAME_PATTERN_PARAMETER_NAME);

        Catalog catalog = (Catalog)((IBmObject)object).bmGetTopObject();
        if (excludeNamePattern != null && catalog.getName().matches(excludeNamePattern))
        {
            return;
        }

        if (object instanceof CatalogAttribute)
        {
            CatalogAttribute attribute = (CatalogAttribute)object;

            if (attribute.getName().startsWith("_")) //$NON-NLS-1$
            {
                String msg = String.format("     %s", attribute.getName()); //$NON-NLS-1$
                resultBuilder.addIssue(msg);
            }
```

```
        }
        else
        {
            TabularSectionAttribute attribute = (TabularSectionAttribute)object;

            if (attribute.getName().startsWith("_")) //$NON-NLS-1$
            {
                String msg = MessageFormat.format("       %s", //$NON-NLS-1$
                    attribute.getName());
                resultBuilder.addIssue(msg);
            }
        }
    }
...
```

Пройдем по новым элементам решения:

- Фильтрация по паттерну имени (наш параметр) применяется не к реквизиту (справочника или табличной части справочника). Однако в качестве входа в проверку будут подаваться только реквизиты (это то, что мы указали в конфигурации проверки). Для того, чтобы обеспечить получение справочника-владельца, мы воспользуемся возможностями БМ по получению объекта верхнего уровня из любого подчиненного объекта (**IBmObject#bmGetTopObject()**). Этот объект, также как и реквизит, автоматически подключается к текущей транзакции, его наличие и актуальность гарантирована (так как реквизит не может существовать без объекта владельца)
- 1C:EDT гарантирует, что в проверку будут поданы только те объекты, которые соответствуют заданной конфигурации. Это означает, что для выяснения, какой именно объект пришел на проверку (реквизит справочника или реквизит ТЧ), достаточно просто проверить его тип

В остальном логика проверки полностью аналогична предыдущим примерам.

3. Проверки внешних свойств (форм и т.д.)

Кроме проверки данных объектов метаданных и общих объектов, 1C:EDT, разумеется, поддерживает проверки и других объектов конфигурации, в т.ч. и внешних свойств, таких как:

- Управляемые формы
- Макеты (СКД, табличные документы и т.п.)

Каждый из этих объектов также является объектом верхнего уровня модели БМ, поэтому принципы создания проверок для них полностью аналогичны.

Тем не менее, мы сможем рассмотреть несколько специфичных сценариев, применимых на данных объектах.

3.1. Пример 4: Проверка имен элементов формы

В отличии от ранее рассмотренных примеров, у элементов формы есть определенная особенность - ее элементы имеют произвольный уровень вложенности.

К счастью, стандартная реализация проверки BasicCheck поддерживает полный обход дерева объектов с произвольной вложенностью.

Определим логику нашей проверки:

- Нам необходимо обойти все элементы формы
- Для каждого элемента нам нужно получить его имя и проверить на наличие недопустимых символов
- В случае обнаружения проблемы - сообщить о ней, привязав сообщение к проблемному элементу

3.1.1. Особенности моделей внешних свойств (и форм - в частности)

С точки зрения модели данных 1C EDT, внешнее свойство представлено в виде объекта модели верхнего уровня (таким же, как и справочник и т.п.). При этом есть определенная специфика:

- Форма имеет уникальное имя (FQN), по которому к ней можно обращаться в модели, при этом FQN формы обязательно содержит в себе имя ее владельца (именно за счет этого обеспечивается контракт FQN)
- Для удобства разработчиков форма связана с объектом-владельцем через соответствующее ссылочное поле (метод доступа - getMdForm()). Данное поле может использоваться для перехода на владельца (например - для фильтрации)

3.1.2. Создаем новую проверку

Создадим новую проверку **FormElementNameCheck**:

```
public class FormElementNameCheck
    extends BasicCheck
{
    public static final String CHECK_ID = "form-element-name-check"; //$NON-NLS-1$

    //  ,
    private static final String EXCLUDE_NAME_PATTERN_PARAMETER_NAME = "excludeNamePattern"; //$NON-NLS-1$

    @Override
    protected void configureCheck(CheckConfigurer builder)
    {
        //@formatter:off
        builder.
            topObject(FORM).
                containment(FORM_FIELD).
                    features(NAMED_ELEMENT__NAME).
            parameter(EXCLUDE_NAME_PATTERN_PARAMETER_NAME, String.class, "", Messages.
FormElementNameCheck_ExcludePattern_Title); //$NON-NLS-1$
        //@formatter:on
    }

    @Override
    protected void check(EObject object, ResultAcceptor resultBuilder, ICheckParameters parameters,
        IProgressMonitor progressMonitor)
    {
        String excludeNamePattern = parameters.getString(EXCLUDE_NAME_PATTERN_PARAMETER_NAME);

        FormField formField = (FormField)object;
        String formFieldName = formField.getName();

        if (excludeNamePattern != null && formFieldName.matches(excludeNamePattern))
        {
            return;
        }

        if (formFieldName.startsWith("_")) //$NON-NLS-1$
        {
            String msg = String.format("     %s", formFieldName); //$NON-NLS-1$
            resultBuilder.addIssue(msg);
        }
    }

    @Override
    public String getCheckId()
    {
        return CHECK_ID;
    }
}
```

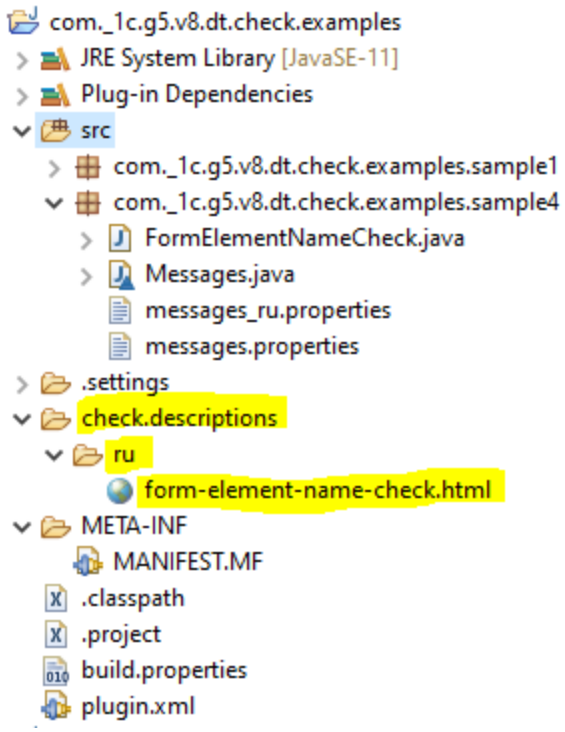
Как мы можем видеть, проверка имеет полностью аналогичную предыдущим структуру

3.1.3. Добавляем внешний файл описания

Многие проверки не являются тривиальными с точки зрения использования и трактовки результатов. Внесение большого объема описания в виде локализуемых строк в Eclipse - задача нетривиальная, поэтому механизм проверок предлагает альтернативный подход для больших описаний проверок - размещение их в виде html-файлов в специфичной файловой структуре. Добавим подобное описание для нашей проверки для демонстрации. Для этого нам понадобится:

- Создать ресурсный каталог с именем **check.descriptions** (имя каталога с точкой, а не вложенный подкаталог) в корневом каталоге нашего бандла с проверками. Обратите внимание, что каталог должен находится в бандле, который декларирует проверки в **plugin.xml** (для случаев, когда структура бандлов, поставляющих проверки, нетривиальна, например когда класс проверки находится в другом бандле), либо в его фрагменте (если есть необходимость отделить описательные данные структурно)
- Создать в нем подкаталог, соответствующий целевому языку. Имя подкаталога должно соответствовать коду языка, возвращаемому функцией java.util.Locale#getLanguage() (коды из подмножества стандартов ISO 639
- Создать файл описания, имеющий имя <**check_id**>.html, где <**check_id**> - идентификатор соответствующей проверки (с учетом case-а)

В результате у нас должна получиться следующая структура (для русского языка, например):



Подсистема проверок 1С EDT при выборе описания руководствуется следующим алгоритмом:

- В случае, если указан файл, соответствующий текущей локали 1С EDT - будет использована данное описание
- Если специфичного для локали описания нет, но есть файл описания по умолчанию (размещенный в корне подкаталога **check.descriptions**) - будет использоваться данное описание
- Если ни одного подходящего файла нет - будет использоваться описание, заданное в самой проверке

Ограничения на файлы описаний:

- Не поддерживаются внешние ресурсы (такие как внешние файлы CSS, картинки и т.п.). При необходимости вставки изображений - используйте embedded-варианты.
- Не гарантируется корректная работа встроенного браузера при переходе на внешние ссылки, поэтому подобные переходы нежелательны

3.2. Пример 5: Проверка табличного документа

В общем случае, проверка табличного документа мало чем отличается от проверки любого объекта метаданных, в первую очередь потому, что внутренняя структура табличного документа не является древовидной/графоподобной. Это означает, что можно описать интересующие объекты и их свойства посредством возможностей стандартной реализации **BasicCheck**.

Однако, есть и ряд особенностей, которые усложняют проверки, а именно:

- Табличные документы могут иметь (потенциально) очень большие размеры. Это означает, что написание проверок "от элементов" (когда в проверке вызываются отдельные элементы, а не весь табличный документ за один раз, с последующим обходом элементов) при операциях полной проверки (импорт проекта, внешнее изменение файла) будут крайне неэффективными). При этом проверка при редактировании - наоборот эффективнее при проверке отдельных элементов
- Элементы табличного документа не адресуются уникальным образом (например, в виде URI), поэтому в настоящий момент позиционирование маркеров на проблемных элементах не реализовано

Для упрощения разработки простых проверок (основанных на **BasicCheck**) в рамках 1С:EDT принято следующее решение:

- Разработчик всегда явно указывает те объекты и свойства, с которыми работает проверка
- Данные указания используются и для определения факта появления нового объекта, и для определения ситуаций с редактированием свойств. Это означает, что разработчик, которому нужно отслеживать изменения в ячейках, должен сконфигурировать проверку для проверки ячеек, и именно их он и получит в качестве входных объектов при проверке, вне зависимости от того, была ли вызвана проверка импортом нового табличного документа, или, например, редактированием одной ячейки
- Это позволяет разработчику не беспокоиться об оптимальных стратегиях обработки данных при реализации простых проверок

4. Проверки языковых модулей

Пример X: Проверка модулей на пустые процедуры (**в разработке**):

```
/**
 * Copyright (C) 2021, 1C
 */
package com.e1c.g5.v8.dt.internal.bsl.check.checks;

import static com._1c.g5.v8.dt.bsl.model.BslPackage.Literals.MODULE;

import java.text.MessageFormat;
import java.util.Set;
import java.util.function.Predicate;
import java.util.stream.Collectors;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.resource.Resource;
import org.eclipse.emf.ecore.util.EcoreUtil;
import org.eclipse.xtext.builder.MonitorBasedCancelIndicator;
import org.eclipse.xtext.resource.IResourceDescription;

import com._1c.g5.v8.dt.bsl.common.IModuleExtensionService;
import com._1c.g5.v8.dt.bsl.common.IModuleExtensionServiceProvider;
import com._1c.g5.v8.dt.bsl.model.Method;
import com._1c.g5.v8.dt.bsl.model.Module;
import com._1c.g5.v8.dt.bsl.resource.BslResource;
import com._1c.g5.v8.dt.bsl.resource.BslResourceDescription;
import com._1c.g5.v8.dt.check.ICheckParameters;
import com._1c.g5.v8.dt.check.components.BasicCheck;
import com._1c.g5.v8.dt.check.settings.IssueType;
import com._1c.g5.v8.dt.common.StringUtils;
import com._1c.g5.v8.dt.mcore.McorePackage;
import com.google.common.collect.Lists;
import com.google.inject.Inject;

/**
 * Unused module method check.
 */
public class ModuleUnusedMethodCheck
    extends BasicCheck
{
    /**
     * ModuleUnusedMethodCheck Id
     */
    public static final String CHECK_ID = "moduleUnusedMethodCheck"; //$NON-NLS-1$

    private static final String EXCLUDE_METHOD_NAME_PATTERN_PARAMETER_NAME =
"excludeModuleMethodNamePattern"; //$NON-NLS-1$

    @Inject
    private IResourceDescription.Manager resourceDescriptionManager;

    @Override
    public String getCheckId()
    {
        return CHECK_ID;
    }

    @Override
    protected void configureCheck(CheckConfigurer builder)
    {
        builder.title(Messages.ModuleUnusedMethodCheck_Title)
            .description(Messages.ModuleUnusedMethodCheck_Description)
            .extension(new ModuleTopObjectNameFilterExtension())
            .parameter(EXCLUDE_METHOD_NAME_PATTERN_PARAMETER_NAME, String.class, StringUtils.EMPTY,
                Messages.ModuleUnusedMethodCheck_Exclude_method_name_pattern_title)
            .issueType(IssueType.WARNING)
```

```
            .module()
            .checkedObjectType(MODULE);
    }

    @Override
    protected void check(EObject object, ResultAcceptor resultAceptor, ICheckParameters parameters,
        IProgressMonitor progressMonitor)
    {
        Module module = (Module)object;

        IModuleExtensionService service = IModuleExtensionServiceProvider.INSTANCE.
getModuleExtensionService();
        String excludeNamePattern = parameters.getString(EXCLUDE_METHOD_NAME_PATTERN_PARAMETER_NAME);

        Predicate<? super Method> predicate = method -> !method.isUsed() && !method.isExport() && !method.
isEvent()
            && service.getSourceMethodNames(method).isEmpty() && !isExcludeName(method.getName(),
excludeNamePattern);
        if (!((BslResource)module.eResource()).isOnlyMethodReparse())
        {
            Set<URI> usedMethods = getUsedMethods(progressMonitor, module.eResource());
            predicate = predicate.and(method -> !usedMethods.contains(EcoreUtil.getURI((EObject)method)));
        }

        module.allMethods()
            .stream()
            .filter(predicate)
            .forEach(unusedMethod -> resultAceptor.addIssue(
                MessageFormat.format(Messages.ModuleUnusedMethodCheck_Unused_method__0, unusedMethod.
getName()),
                unusedMethod, McorePackage.Literals.NAMED_ELEMENT__NAME));
    }

    private Set<URI> getUsedMethods(IProgressMonitor progressMonitor, Resource resource)
    {
        IResourceDescription descr = resourceDescriptionManager.getResourceDescription(resource);
        return (descr instanceof BslResourceDescription
            ? Lists.newArrayList(((BslResourceDescription)descr)
                .getReferenceDescriptions(new MonitorBasedCancelIndicator(progressMonitor)))
            : Lists.newArrayList(descr.getReferenceDescriptions())).stream()
                .map(reference -> reference.getTargetEObjectUri())
                .collect(Collectors.toSet());
    }

    private boolean isExcludeName(String name, String excludeNamePattern)
    {
        return StringUtils.isNotEmpty(excludeNamePattern) && name.matches(excludeNamePattern);
    }
}
```

5. Комплексные сценарии проверки

Пример X: Проверка объекта на вхождение в состав подсистемы

- Комплексная проверка с шедулингом проверки для не-целевого объекта

Пример X: Проверка обработчика события формы

- Совмещенная проверка модуля и модели

Пример X: Проверка QL

В настоящий момент система проверок 1C:EDT не поддерживает проверки запросов нативным образом (нативная поддержка будет представлена в следующих версиях). Однако, существует возможность валидации запросов с ограниченными возможностями по привязке маркеров, а именно:

- Маркеры могут привязываться к определенному месту в запросе, но это место не транслируется системой переходов от маркеров в редакторы и не отображаются в редакторах запросов (т.е. доступно только в панели проблем конфигурации)
- Валидация запросов в формах, СКД и модулях делается отдельно, со своей спецификой для каждого случая

Рассмотрим данные особенности на практике:

Документ в разработке

6. CLI

7. Тестирование

8. Приложения

8.1. Соглашения при написании проверок

```
# Check convention -


##

###

 1:         1C:EDT.
        ,    .

###

1. Cebab-case (-) -   ,  ,    .    `CamelCase` ..   .
2.    (md, form, right, ql, dcs  .) -           (common-module, catalog, role  ..)
3.   (v8std)
4.       ,    Quick-fix

###


 ,
,


###

1.
2.     /ru/-.md
3.
4.      ""  ""

###  (issueType)



- `ERROR` -
- `WARNING` -
- `SECURITY` -
- `PERFORMANCE` -
- `PORTABILITY` - ?
- `LIBRARY_DEVELOPMENT_AND_USAGE` -        .
- `CODE_STYLE` -    ,
- `UI_STYLE` -   UI
- `SPELLING` -   , ,

###  (severity)

      .
        .
```

- `BLOCKER` -
- `CRITICAL` - ,
- `MAJOR` - ,
- `MINOR` -
- `TRIVIAL` -

(complexity)

- `NORMAL` -   ,   .
- `COMPLEX` -   ,

###

  :
1.
2.   - =
3.
4.   (, ,   ,)

 :
1.
2. , ,

   :
1.
2.   `CamelCase`,
3.

##

   ,

###

1.
2. : ,
2.

:
1.
2.
3.

###

1.
2. ,
3. ,

  ,   - ,   .

##

1.   ,
2.   ,   .

          "" ""

##

 -     - .
     -           .

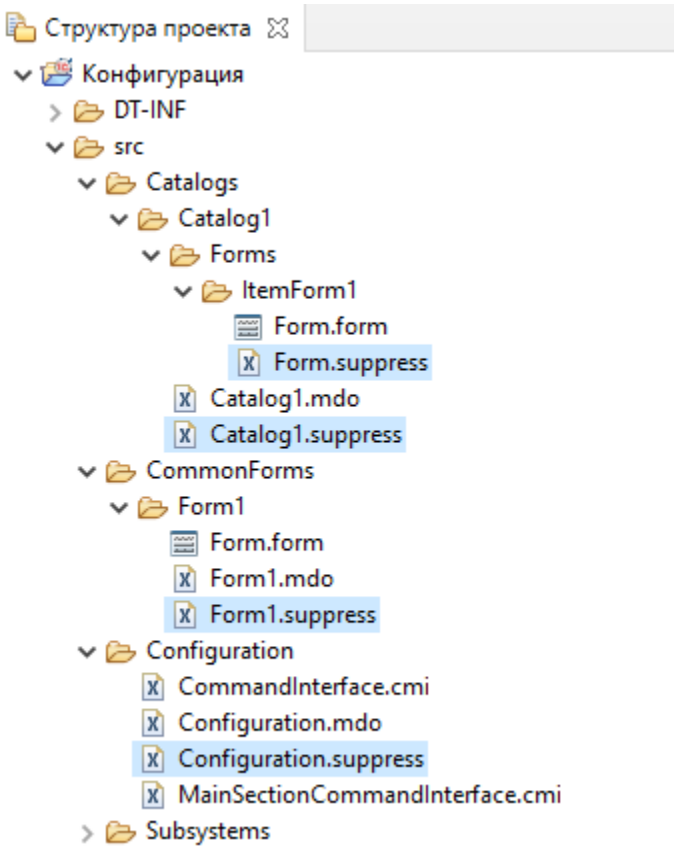
 ,   ,   (issue) .

8.2. Формат файлов настроек подавления ошибок

Настройки подавления ошибок представляют собой набор ресурсных файлов XML с расширением ***.suppress**, поставляемых вместе с файлами конфигурации.

Файлы настроек располагаются в каталогах объектов рядом с mdo-файлами либо во вложенных каталогах. Имена файлов настроек подавления коррелируют с именами объектов.

Например, для объекта **Configuration** можно создать файл настроек **Configuration.suppress**, для объекта справочник **Catalog1**, соответственно **Catalog1.suppress** и т.д.



Общий принцип размещения и именования файлов:

- Для объектов метаданных и общих объектов имя файла настроек совпадает с именем файла объекта (***.mdo**)
- Для внешних свойств имя файла настроек совпадает с именем файла внешних настроек (например для форм это будет "**Form.suppress**")
- В некоторых случаях (таких как общие формы и т.п.) несколько файлов настройки могут находиться в одном каталоге (например настройки подавления для объекта метаданных "Общая форма" и собственно управляемой формы, соответствующей объекту метаданных)

8.2.1. Структура и виды настроек подавления сообщений

suppress-файлы для различных видов объектов конфигурации имеют одинаковую структуру:

- **root-секция** - имя класса подавления
- Одна или несколько **suppression-секций** для описания подавления
- **containment-секция**, которая внутри себя тоже может содержать **suppression-секции**. containment-секции есть не у всех видов настроек.
- **method-секция**, которая внутри себя тоже может содержать **suppression-секции**. Есть только у настроек bsl-модулей.

Остановимся подробнее на каждой из секций.

8.2.1.1.1. root-секция

Есть в каждом файле настроек. По сути это корневой тэг с именем класса настройки и дополнительными обязательными и необязательными атрибутами.

fqn - обязательный атрибут для всех root-секций, кроме настроек конфигурации. Как следует из названия, это fully qualified name объекта, к которому относятся настройки.

suppressed - необязательный атрибут (по умолчанию, false) булевого типа. Если **suppressed=true**, то все сообщения, относящиеся непосредственно к этому объекту, будут подавлены. При этом, в нижележащих секциях этот параметр может быть переопределен. Здесь действует правило - чем глубже по уровню находится настройка подавления, тем большим приоритетом она обладает.

8.2.1.1.2. suppression-секция

Это описание подавления в формате ключ-значение. Ключом является check ID проверки, а значение - булевская величина для управления подавлением проверки. Например: **<suppressions key="TestCheckId1" value="true"/>**

При этом, как отмечалось выше, если в родительской секции значение suppressed было выставлено, то использоваться для конкретной проверки будет значение подавления из suppression-секции, которое в свою очередь может быть переопределено в дочерних секциях (containment или method).

8.2.1.1.3. containment-секция

Эта секция, так же как и root-секция, является контейнером для **suppression-секций** и для других containment-секций. У секции есть обязательный атрибут **fqn** (смысл его тот же, что и для root-секции) и необязательный атрибут **suppressed**.

8.2.1.1.4. method-секция

Аналогична containment-секции, но используется для настройки подавления сообщений, относящимся к методам bsl-модулей.

Всего поддерживается **4** вида файлов настроек подавления:

1. SuppressConfiguration
2. SuppressSubsystem
3. SuppressModule
4. SuppressGenericObject

Первые три - для объектов, соответствующих названию класса подавления (Configuration, Subsystem, Module), и последний (GenericObject) для всех остальных объектов.

8.2.2. Примеры файлов настроек подавления сообщений

8.2.2.1. Файл настроек для подавления сообщений на уровне конфигурации (Configuration.suppress)

Configuration.suppress

```
<?xml version="1.0" encoding="UTF-8"?>
<suppress:SuppressConfiguration xmlns:suppress="http://g5.1c.ru/v8/dt/check/suppress/model" suppressed="true"
>
    <suppressions name="TestCheckId1" suppressed="false"/>
</suppress:SuppressConfiguration>
```

Замечание: В данном примере проверка *TestCheckId1* не будет подавлена, т.к. она переопределена на уровне suppression-секции. Все остальные проверки для конфигурации будут подавлены. Так же проверки *TestCheckId1* будут подавлены и для всех остальных объектов, если для них не будут созданы переопределения на уровне объектов.

8.2.2.2. Файл настроек для подавления сообщений на уровне общей формы, ее атрибутов, параметров, команд и элементов (Form1.suppress)

Form1.suppress

```
<suppress:SuppressGenericObject xmlns:suppress="http://g5.1c.ru/v8/dt/check/suppress/model" fqn="CommonForm.
Form1.Form" suppressed="false">
   <suppressions key="TestCheckId1" value="true"/>
   <suppressions key="TestCheckId2" value="false"/>

   <containments fqn="attributes:Attr1" suppressed="false">
      <suppressions key="TestCheckId1" value="false"/>
      <suppressions key="TestCheckId2" value="true"/>
   </containments>

   <containments fqn="attributes:Attr2" suppressed="true">
```

```
      <suppressions key="TestCheckId1" value="false"/>
   </containments>

   <containments fqn="items:Field1">
      <suppressions key="TestCheckId1" value="false"/>
      <suppressions key="TestCheckId2" value="true"/>
   </containments>

   <containments fqn="parameters:Param1">
      <suppressions key="TestCheckId1" value="false"/>
      <suppressions key="TestCheckId2" value="true"/>
   </containments>

   <containments fqn="formCommands:Command1">
      <suppressions key="TestCheckId1" value="false"/>
      <suppressions key="TestCheckId2" value="true"/>
   </containments>
</suppress:SuppressGenericObject>
```

8.2.2.3. Файл настроек для подавления сообщений на уровне справочника (Catalog.suppress)

Catalog.suppress

```
<?xml version="1.0" encoding="UTF-8"?>
<suppress:SuppressGenericObject xmlns:suppress="http://g5.1c.ru/v8/dt/check/suppress/model" fqn="Catalog.
Catalog" suppressed="true">
    <suppressions key="Catalog_CheckId" value="false"/>
</suppress:SuppressGenericObject>
```

8.2.2.4. Файл настроек для подавления сообщений на уровне модуля справочника (Catalog. ManagerModule.suppress)

Catalog.ManagerModule.suppress

```
<?xml version="1.0" encoding="UTF-8"?>
<suppress:SuppressModule xmlns:suppress="http://g5.1c.ru/v8/dt/check/suppress/model" fqn="Catalogs.Catalog.
ManagerModule" suppressed="true">
    <suppressions key="TestCheckId1" value="false"/>
</suppress:SuppressModule>
```

Замечание: Обратите внимание на составное имя файла настроек подавлений и fqn.

8.2.2.5. Файл настроек для подавления сообщений на уровне справочника, его атрибутов, табличных частей и атрибутов табличных частей (CatalogWithAttributesAndTabularSections.suppress

CatalogWithAttributesAndTabularSections.suppress

```
<?xml version="1.0" encoding="UTF-8"?>
<suppress:SuppressGenericObject xmlns:suppress="http://g5.1c.ru/v8/dt/check/suppress/model" fqn="Catalog.
CatalogWithAttributesAndTabularSections" suppressed="false">
    <suppressions key="Suppressed_TestCheckId1" value="true"/>
    <containments fqn="attributes:Attribute1" suppressed="true">
        <suppressions key="TestCheckId1" value="false"/>
    </containments>
    <containments fqn="tabularSections:TabularSection1" suppressed="false">
        <suppressions key="TestCheckId1" value="false"/>
        <suppressions key="TabularSection_TestCheckId1" value="true"/>
        <suppressions key="SuppressedTabularSection_TestCheckId1" value="true"/>
        <containments fqn="attributes:TabularSection1Attribute2" suppressed="false">
            <suppressions key="TestCheckId1" value="false"/>
            <suppressions key="TabularSection_TestCheckId1" value="false"/>
            <suppressions key="TabularSectionAttribute_TestCheckId1" value="false"/>
            <suppressions key="SuppressedTabularSectionAttribute_TestCheckId1" value="true"/>
        </containments>
```

```
    </containments>
</suppress:SuppressGenericObject>
```

Замечание: Для настройки подавления сообщений, относящихся к табличным частям и атрибутам используются вложенные containment-секции.

8.2.2.6. Файл настроек для подавления сообщений на уровне подсистемы третьего уровня вложенности (Subsystem111.suppress)

Subsystem111.suppress

```
<?xml version="1.0" encoding="UTF-8"?>
<suppress:SuppressSubsystem xmlns:suppress="http://g5.1c.ru/v8/dt/check/suppress/model" fqn="Subsystem.
Subsystem1.Subsystem.Subsystem11.Subsystem.Subsystem111" suppressed="true">
   <suppressions key="TestCheckId1" value="false"/>
</suppress:SuppressSubsystem>
```

Замечание: Для подсистем тоже действуют правила наследования и переопределения подавлений, но уже на уровне иерархии объектов подсистем.

8.2.2.7. Файл настроек для подавления сообщений на уровне модуля и методов (ModuleWithMethods.suppress)

ModuleWithMethods.suppress

```
<?xml version="1.0" encoding="UTF-8"?>
<suppress:SuppressModule xmlns:suppress="http://g5.1c.ru/v8/dt/check/suppress/model" fqn="CommonModules.
ModuleWithMethods.Module" suppressed="false">
    <suppressions key="TestCheckId1" value="false"/>
    <methods name="TestProcedure" suppressed="false">
        <suppressions key="TestCheckId1" value="true"/>
    </methods>
</suppress:SuppressModule>
```