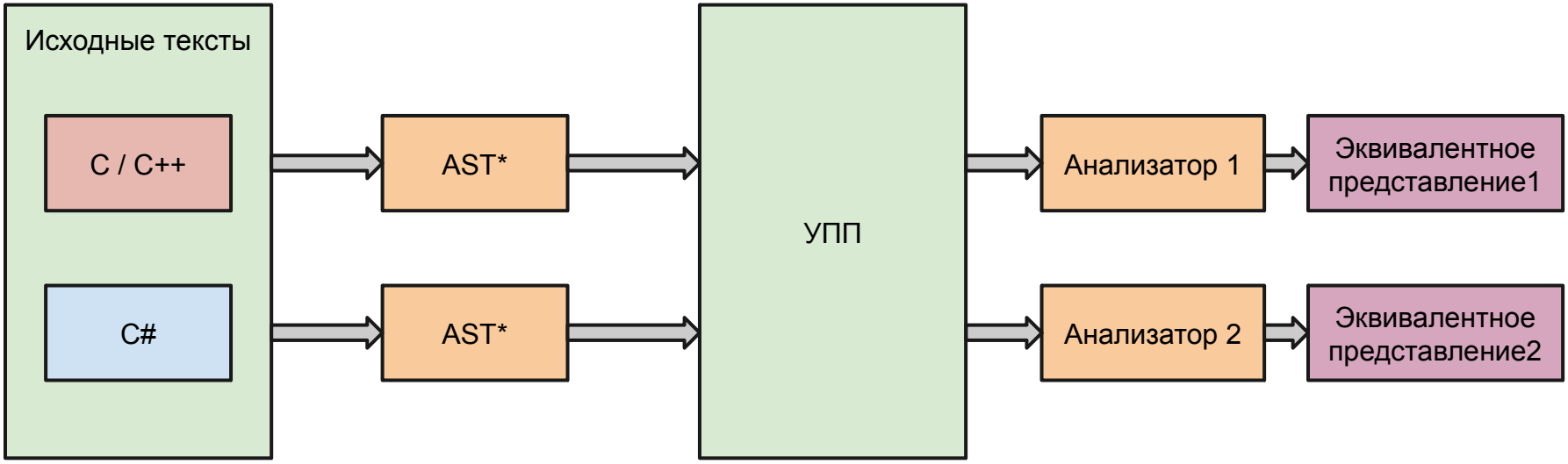
Построение эквивалентного представления зависимостей в исходном тексте программ с использованием универсального промежуточного представления

Пустыгин А.Н., Ошнуров Н.А., Ковалевский А.А.
Челябинский Государственный Университет

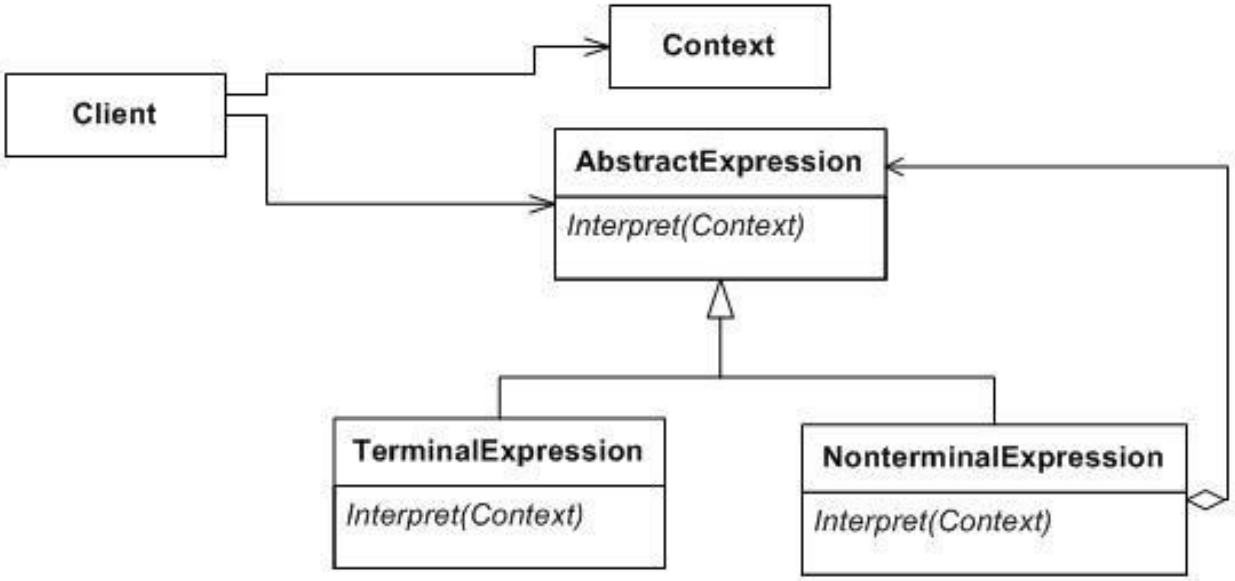
Универсальное промежуточное представление



Представление зависимостей

Граф зависимостей в исходном тексте— ориентированный граф, отображающий соотношение элементов (сущностей) текста некоторой совокупности в соответствии с выбранным транзитивным отношением над ней.

Пример графа зависимостей в исходном тексте



Исходный текст

```
class Context { };
class AbstractExpression {
public:
virtual void Inperpret(Context ctx) = 0;
};
. . . . . .
class TerminalExpression : public AbstractExpression {
virtual void Inperpret(Context ctx) { }
};
. . . . . .
class NonterminalExpression : public AbstractExpression {
virtual void Inperpret(Context ctx) { }
AbstractExpression* _children;
};
. . . . . .
class Client {
Context    _ctx;
AbstractExpression* _expr;
}
```

Задача

Построение эквивалентного представления зависимостей в исходном тексте.

Задача

Построение эквивалентного представления зависимостей в исходном тексте на основе УПП универсального промежуточного представления.

Преимущества УПП:

- Содержит в себе всю информацию об исходных текстах
- Позволяет применять единый инструмент для построения представления зависимостей вне зависимости от используемых языков программирования

Сущности исходного текста, между которыми строятся зависимости

1. Пользовательские типы(классы, интерфейсы)
2. Операции над данными (методы)
3. Объединения типов (пространства имен, прообразы паттернов)
4. Логические единицы, содержащие скомпилированный код (исполняемые файлы, разделяемые библиотеки)

Упрощенная модель УПП, используемая при построении зависимостей

Множество всех аргументов методов

$$A = \{a_i \in N \mid Type_{a_i} = "Arg" \}$$

Множество всех методов

$$M = \{m_i \in N \mid Type_{m_i} = "MethodDecl" \}$$

Множество всех пользовательских типов

$$C = \{c_i \in N \mid Type_{c_i} = "Class" \}$$

Типы связей при описании зависимостей. Наследование

Наследование - механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса.

Модель наследования в УПП.

$$\exists\, c_k, c_{k+1} : Parent_{c_k} = c_p, Type_{c_k} = "Inherits", Type_{c_{k+1}} = "Inherit", Parent_{c_{k+1}} = c_k,$$

Где *Parentn* - родительский узел для n,
Typen - тип узла n

Тип является базовым типом класса, если он представлен в ветке <*Inherits*>/<*Inherit*> определения этого класса.

Фрагмент УПП, отражающий зависимость наследования.

```
<Class id="53458261" name="Nancy.DefaultNancyContextFactory"
kind="class">
    ...
    <Inherits>
            <Inherit type="public">
                <Type ref="1" name="object" kind="internal" />
            </Inherit>
            <Inherit type="public">
                <Type ref="44305076" name="Nancy.INancyContextFactory"
    kind="interface" />
            </Inherit>
        </Inherits>
        ...
    <Class>
```

Класс Nancy.DefaultNancyContextFactory наследуется от object и Nancy.INancyContextFactory

Тип зависимости «агрегация» в УПП.

$$c_a \; descend \; c_c, \; Type_{c_a} = Type, \; \nexists c_k : Type_{c_k} = "Arg", \; c_a \; descend \; c_k$$

Где *p descend l* означает, что *p* потомок *c*

Тип агрегирован в объекте, если ссылка на этот тип представлена внутри определения этого объекта.

Фрагмент УПП, описывающий зависимость «Агрегация».

```
<MethodDecl id="19982180" name="Invoke" kind="method" argc="1"
line="61" pos="8">
    ...
    <Type ref="28400769"
name="System.Threading.CancellationToken" kind="struct" />
    ...
</MethodDecl>
```

Тип System.Threading.CancellationToken агрегируется в методе Invoke.

Тип зависимости «Действие» в УПП

Модель

$m_m \in M\ method\ c_c\ \in C$, если $Parent_{m_m} = C_c$

Пример.

```
<MethodDecl id="56030827" kind="constructor" line="11" pos="8">
   <Block line="12" pos="8">
       ...
   </Block>
</MethodDecl>
```

Типы зависимостей между сущностями в исходном тексте

1. Явная. Связь явно представлена в исходном тексте.
2. Неявная. Связь возможна, но не представлена в исходном тексте.

Пример явной связи в исходном тексте.

```
class BodyPart
{ };

class Human
{
private:
    BodyPart* _bodyParts;
};
```

Явная связь между Human и BodyPart. BodyPart агрегируется в классе Human.

Пример неявной связи в исходном тексте.

```
class IBodyPart { };

class Leg : public IBodyPart { };
class Arm : public IBodyPart { };

class Human
{
private:
    IBodyPart* _bodyParts;
};
```

Неявные связи между Human и Leg, Human и Arm. Leg и Arm являются наследниками агрегированного типа

Алгоритм построения представления зависимостей между сущностями в исходном тексте

Этапы алгоритма:

1. Общий анализ - анализ существующих зависимостей, составление списка всех сущностей, учавствующих в процессе анализа.
2. Анализ неявных зависимостей, среди всех обнаруженных сущностей.
3. Обработка зависимостей, их дополнение, сохранение в файл.

Алгоритм. Этап 1 - Общий анализ

```
Для каждого узла xml   N:
    Если N не является объектом связи, то:
        Продолжить для следующего узла

    Установить все явные связи N (согласно предыдущим правилам)
    Записать информацию о связях в таблицу зависимостей

    Если N является объявлением типа:
        Занести информацию о типе в таблицу типов
        Занести информацию о родителях типа для N
        Пометить тип как текущий

    Если N является объявлением метода:
        Занести информацию о методе N для текущего типа
```

Алгоритм. Этап 2 - Анализ неявных зависимостей

```
Для каждого типа T1 в таблице типов:
    Для каждого типа T2 в таблице типов, такого, что T2 != T1:
        Если множество методов T1 из множества методов T2:
            Пометить T2 как удовлетворяющий интерфейсу T1
```

Два метода считаются равными, если имеют совпадающие типы аргументы и тип возвращаемого значения

Алгоритм. Этап 3 - Обработка зависимостей

Для всех связей ***R*** между типами ***T1*** и ***T2***:

Добавить неявную связь между ***T1*** и всеми типами, удовлетворяющими интерфейсу ***T2***

Добавить неявную связь между ***T1*** и всеми типами, наследниками ***T2***

Пример проблемы исходного текста. Нераспознанный интерфейс

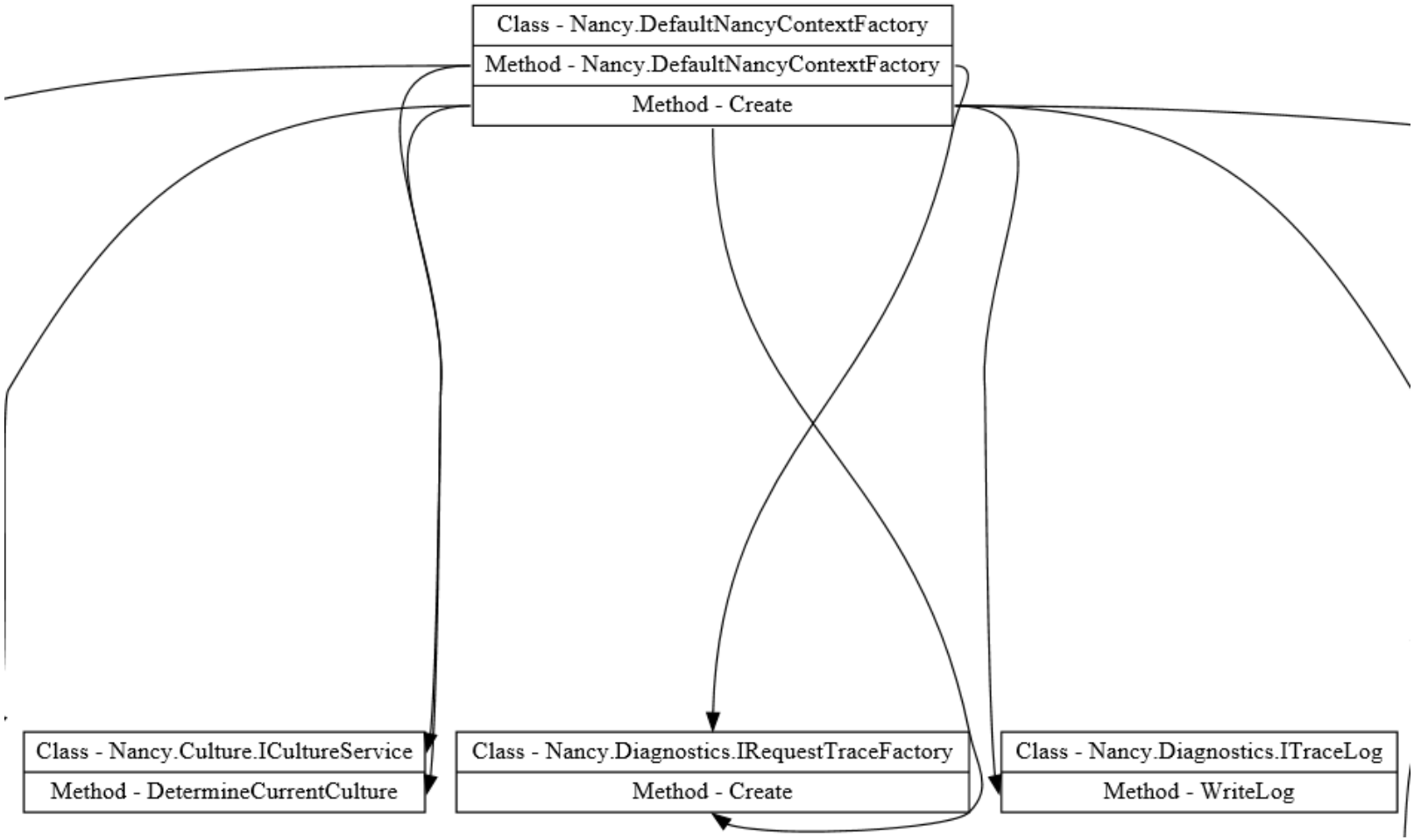
```
class Flying {
void Fly(int altitude) { }
};
. . . . . .
class Bird {
void SitOnTwig() { }
void Fly(int altitude) { }
};
. . . . . .
class Airplane {
void Fly(int altitude) { }
};
```

Классы Bird, Airplane удовлетворяют интерфейсу Flying, но при этом не реализуют его.

Пример представления зависимостей в эквивалентном представлении. Наследование

```
<?xml version="1.0" encoding="utf-8"?>
<dependecy_graph>
    <entities>
        <entity id="48412091" name="Nancy.DefaultNancyContextFactory"
    type="class" />
        <entity id="56030827" name="Nancy.DefaultNancyContextFactory"
    type="method" />
        ...
    </entities>
    <relations>
        <relation from="56030827" to="-278492093" type="aggregation" />
        <relation from="56030827" to="-1466215831" type="call" />
        <relation from="56030827" to="1" type="injection" />
        ...
        <relation from="48412091" to="-1991265721" type="inheritance" />
        <relation from="48412091" to="23580843" type="aggregation" />
        <relation from="48412091" to="56030827" type="member" />
    </relations>
    </dependecy_graph>
```

Визуализация(dot) представления зависимостей. Пример наследования



Результаты анализа зависимостей для тестовых проектов

	NancyFx.Core	Newtonsoft.Json
Количество классов	411	204
Среднее количество методов на класс	4	9
Количество связей типа “Агрегация”	4153	4962
Количество связей типа “Вызов”	2681	2769
Количество связей типа “Передача аргумента при вызове метода”	3134	3137
Количество связей типа “Наследование”	356	196

Ссылки

1. XML схема представления зависимостей
   https://github.com/ifanatic/CodeAnalysis/blob/master/doc/dependencygraph.xsd
2. NancyFx.Core
   https://github.com/NancyFx/Nancy
3. Newtosoft.Json
   https://github.com/JamesNK/Newtonsoft.Json

Спасибо за внимание!