Aleksander Bielawski
Łukasz Panek

Dokumentacja końcowa projektu z przedmiotu ZPR
Zadanie 3 - narzędzie do zmniejszania liczby zależności pomiędzy plikami

Treść zadania

W wybranym katalogu zawierającym kod źródłowy w C/C++ wyszukiwać poleceń preprocesora `#include "nazwa"` i na ich podstawie zbudować graf (skierowany) zależności pomiędzy plikami. Zbudować drzewo rozpinające, aby usunąć nadmiarowe powiązania pomiędzy plikami, generując odpowiednie komunikaty, np. *W pliku "nazwaA" można usunąć zależność od pliku "nazwaB"*. Sugestia: wykorzystać bibliotekę Boost Graph Library.

Postać rozwiązania

Program posiada interfejs tekstowy. Katalog z projektem zadawany jest jako argument wywołania programu. Katalog taki jest rekursywnie przeglądany w poszukiwaniu plików źródłowych języka C++ (i C). Rodzaj plików rozpoznawany jest poprzez dopasowanie rozszerzenia do określonych wzorców wymienionych w części dokumentacji dotyczącej założeń. Program akceptuje takie projekty, których struktura umożliwia wykonanie założonego zadania (znalezienie drzewa rozpinającego). W przypadku gdy struktura taka nie jest, użytkownik informowany jest o rodzaju błędu (np. o cyklicznej zależności plików).

Program, nazwany *deptool* (od `dependency tool`), wywoływany jest w następujący sposób:

```
deptool <nazwa katalogu z projektem> [rozszerzenia, np. .impl]
```

Natomiast oczekiwane wyjście jest albo stosownym komunikatem w przypadku niepoprawności struktury projektu, albo listą komunikatów następującej postaci:

```
Below are listed all unnecessary dependencies between files of your project:
nazwaA.cpp including nazwaB.hpp.
nazwaA.cpp including nazwaC.hpp.
```

Przenośność projektu

Aplikacja została przygotowana do poprawnej kompilacji i działania w środowiskach Windows i Linux. Kod jest w całości identyczny dla obu platform: przy implementacji nie zostały wykorzystane żadne specyficzne, dostępne wyłącznie dla określonego środowiska biblioteki lub techniki. Źródła aplikacji dostosowano do kompilacji pozbawionej ostrzeżeń z użyciem kompilatorów Visual C++, g++, a także clang++. Proces budowy dla obu środowisk został zautomatyzowany poprzez zastosowanie narzędzia Scons.

Metody i stopień realizacji

Wynikowa aplikacja realizuje wszystkie założenia odnośnie jej działania:

Interfejs tekstowy

Metoda realizacji

Utworzono klasę o interfejsie dostosowanym do przekazywania komunikatów użytkownikowi oraz przetwarzania argumentów wywołania programu.

Przegląd katalogu w poszukiwani plików źródłowych

Metoda realizacji

Utworzono klasy reprezentujące istotne z punktu widzenia aplikacji obiekty systemu plików: katalog zadanego projektu oraz pliki stanowiące jego zawartość.

Nietrywialne techniki i rozwiązania programistyczne

•Użyto mechanizmu obsługi wyjątków w celu zapewnienia reakcji na błędy spowodowane np. brakiem wskazanego katalogu z projektem, brakiem dostępu, dysfunkcjonalnością systemu plików.

•Użyto mechanizmu szablonów w celu utworzenia reprezentacji pliku źródłowego parametryzowanej typem obiektu parsującego jego zawartość.

•Wykorzystano mechanizm metod wirtualnych w celu zastosowania wzorca projektowego Wizytator, użytego do stworzenia obiektu wizytującego hierarchię katalogu projektu i wyszukującego w nim plików źródłowych.

•Użyto wzorca projektowego Fabryka do tworzenia instancji klas reprezentujących pliki na podstawie ich ścieżek w systemie plików.

•Klasy związane z systemem plików rozmieszczone zostały w stosownej hierarchii klas.

•Wykorzystano sprytne wskaźniki w celu uniknięcia wielokrotnego kopiowania potencjalnie dużych kolekcji znalezionych plików źródłowych.

•Implementacja klas reprezentujących obiekty systemu plików oparta została o funkcjonalności dostarczane przez boost::filesystem, w szczególności path i directory_recursive_iterator. Wykorzystano także boost::bind oraz boost::transform_iterator w celu zachowania zwięzłości i sprawnego przetwarzania kolekcji obiektów reprezentujących pliki.

Parsowanie plików źródłowych

Metoda realizacji

Utworzono klasę parsera udostępniającą funkcjonalność wydobywania informacji o zawartości dyrektyw #include znajdujących się w zadanym pliku źródłowym.

Nietrywialne techniki i rozwiązania programistyczne

•Do implementacji klas parsera użyto funkcjonalności dostarczanych przez boost::regex, w szczególności sregex_iterator. Do znajdowania zawartości dyrektyw #include zastosowano następujące wyrażenie regularne:

```
\#[\s]*include[\s]+"(.+?)"([^\n]*)$
```

•Do wczytywania zawartości pliku do analizy zastosowano należące do biblioteki standardowej iteratory strumieni istreambuf_iterator.

Rozpoznawanie zależności plików

Metoda realizacji

Realizacji dokonano poprzez zawarcie w interfejsie klasy reprezentującej pliki źródłowe możliwości przekazania użytkownikowi klasy kolekcji obiektów reprezentujących pliki źródłowe, od których zależy dany plik. Do implementacji tej części interfejsu wykorzystano parsowanie zawartości pliku źródłowego. Otrzymane w wyniku parsowania informacje stosowane są do lokalizacji załączanych plików w systemie plików.

Nietrywialne techniki i rozwiązania programistyczne

•Użyto mechanizmu obsługi wyjątków w celu zapewnienia reakcji na błędy spowodowane brakującymi zależnościami pliku.

•W implementacji wykorzystano boost::filesystem::fstream w celu dostarczenia parserowi strumienia wejściowego. Użyto też boost::bind oraz boost::transform_iterator w celu zachowania zwięzłości i efektywniejszej konstrukcji kolekcji plików stanowiących zależności.

Budowa grafu zależności

Metoda realizacji

Utworzono klasę grafu, który jako węzły przechowuje obiekty reprezentujące pliki źródłowe projektu, natomiast jako krawędzie – zależności między nimi. Graf ten jest zarządzany przez oddzielną klasę. Klasa zarządzająca buduje graf na podstawie kolekcji wszystkich obiektów reprezentujących pliki źródłowe projektu i zależności między nimi. Przy budowie grafu klasa zarządzająca korzysta z pomocniczej mapy plików źródłowych jako kluczy i ich deskryptorów jako wartości – jest to konieczne, gdyż każdy plik źródłowy w kolekcji plików może wystąpić wielokrotnie (może być włączany przez wiele innych) i należy uniknąć ryzyka utworzenia wielu węzłów grafu reprezentujących ten sam plik projektu.

Nietrywialne techniki i rozwiązania programistyczne

•Użyto mechanizmu wyjątków w celu zgłaszania błędów spowodowanych brakującymi zależnościami pliku.

•Użyto mechanizmu szablonów do tworzonego grafu zależności, parametryzowanego typem obiektu przechowywanego w każdym węźle grafu.

•W implementacji szablonu klasy grafu wykorzystano boost::graph::adjacency_list jako klasy adaptowanej (skorzystano z wzorca projektowego adapter obiektów).

•Wykorzystano STL-owy szablon klasy map w celu implementacji mapy plików źródłowych i ich deskryptorów.

Badanie poprawności struktury projektu

Metoda realizacji

Funkcjonalność zrealizowano, sprawdzając, czy między węzłami grafu nie ma cyklicznych zależności oraz czy graf ten jest spójny (tzn. czy wszystkie pliki źródłowe znalezione w katalogu projektu rzeczywiście się nań składają). Sprawdzenie to dokonywane jest na żądanie klasy zarządzającej grafem.

Nietrywialne techniki i rozwiązania programistyczne

•Użyto mechanizmu obsługi wyjątków w celu zapewnienia reakcji na błędy spowodowane znalezieniem cyklicznych zależności między plikami źródłowymi projektu oraz stwierdzenia niespójności grafu projektu.

•Wykorzystano boost::graph::depth_first_search w celu określenia acykliczności grafu.

•Użyto sprytnych wskaźników w celu uniknięcia wielokrotnego kopiowania kolekcji znalezionych krawędzi cyklicznych.
•W implementacji kolekcji znalezionych krawędzi cyklicznych wykorzystano szablony klas vector i pair z biblioteki STL.

Wyszukiwanie w grafie projektu drzewa rozpinającego

Metoda realizacji

Wyszukiwanie nadmiarowych włączeń między plikami projektu zapisanymi w grafie odbywa się na żądanie klasy nim zarządzającej. W odpowiedzi graf najpierw sortuje topologicznie swoje węzły (tzn. każdy plik projektu zostaje dodany do posortowanej kolekcji dopiero wtedy, gdy znajdą się już w niej wszystkie pliki, od jakich jest zależny). Następnie następuje przejście przez posortowaną kolekcję i dodanie do zbioru plików osiągalnych z danego pliku – oprócz plików bezpośrednio przez ten plik włączanych – także tych plików, które są włączane przez pliki bezpośrednio przez ten plik włączane. Na koniec, dla każdego pliku zapisanego w grafie sprawdzane jest, czy któryś z plików bezpośrednio przez niego włączanych znajduje się w zbiorze plików osiągalnych z innego bezpośrednio przez ten plik włączanego pliku. Jeśli tak, to takie włączenie jest włączeniem nadmiarowym.

Nietrywialne techniki i rozwiązania programistyczne

•Użyto sprytnych wskaźników w celu uniknięcia wielokrotnego kopiowania kolekcji nadmiarowych powiązań między plikami projektu, a także kolekcji topologicznie posortowanych węzłów grafu.
•Do implementacji kolekcji nadmiarowych powiązań między plikami użyto szablonów klas vector i pair z biblioteki STL, do implementacji zbioru plików osiągalnych z danego pliku wykorzystano specjalizację vector<bool> (gdzie indeks wektora pokrywa się z indeksem pliku w grafie, natomiast wartość logiczna odpowiada na pytanie: „Czy plik o tym indeksie jest osiągalny z danego pliku?”).

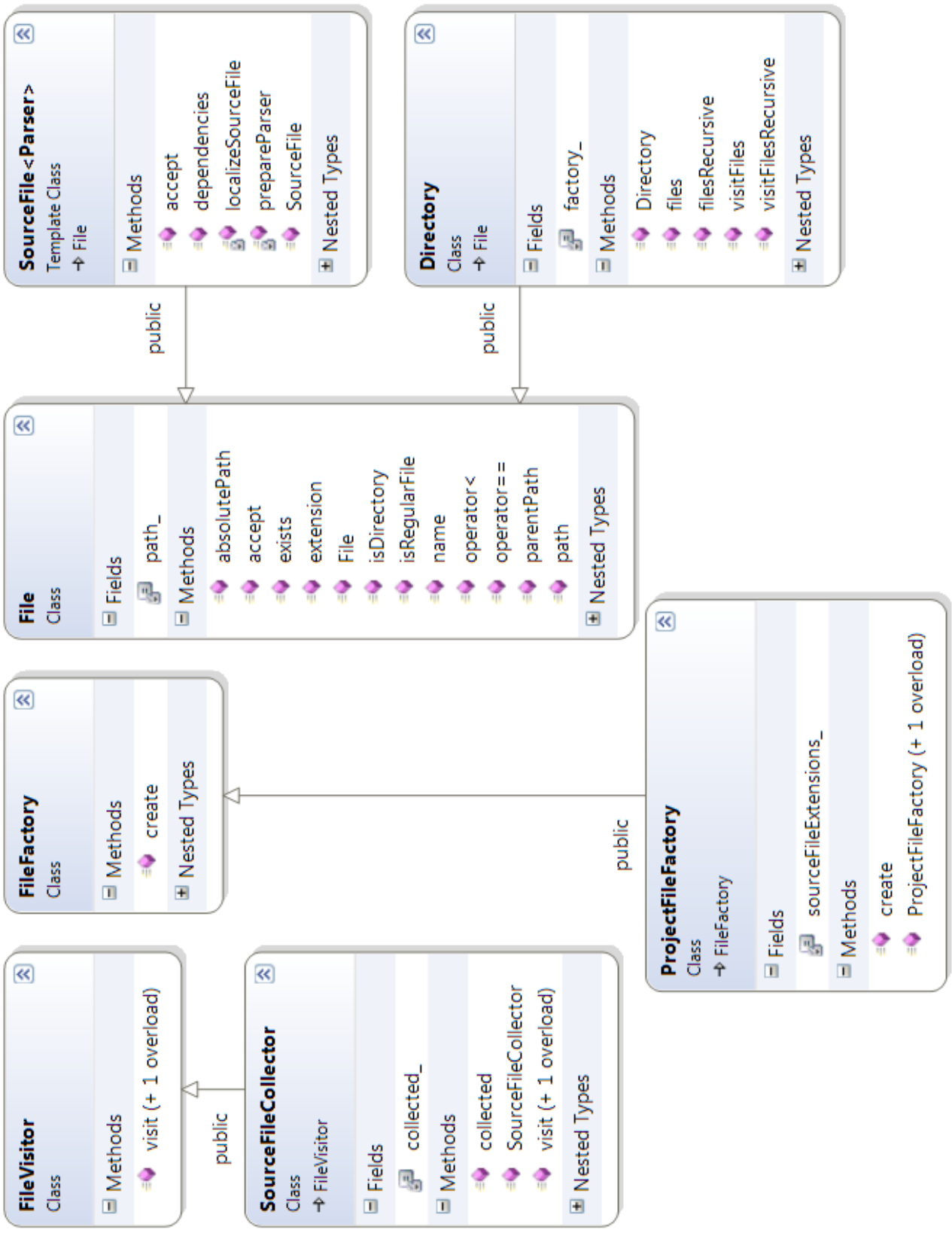
Testy jednostkowe

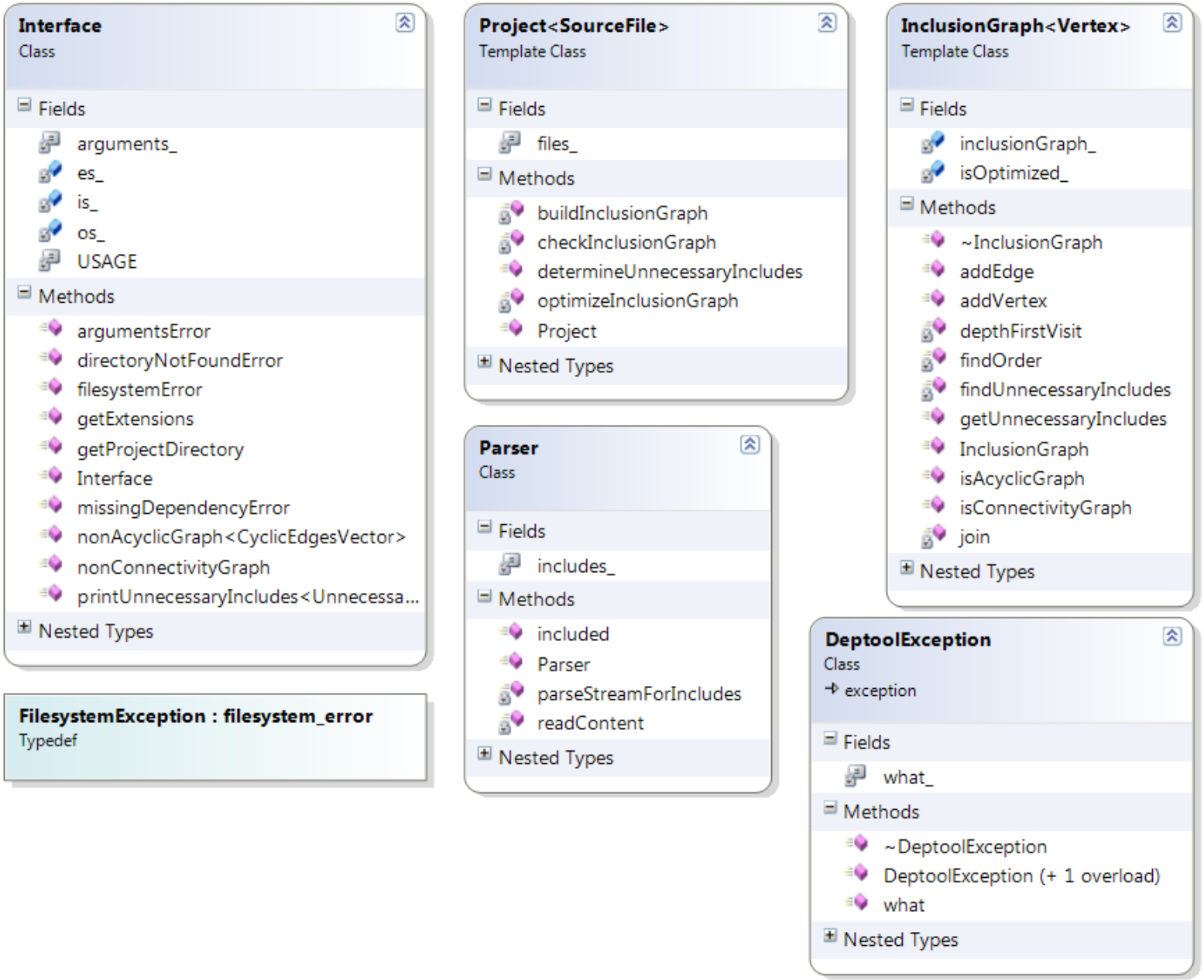
Do każdej dostarczającej nietrywialną funkcjonalność klasy przygotowano testy jednostkowe sprawdzające zgodność każdego elementu interfejsu klasy ze specyfikacją dostarczoną w formacie Doxygen. Budowa testów możliwa jest do wyspecyfikowania w narzędziu SCons jako cel „tests”. Kod testów przystosowany jest do czystej kompilacji na obu docelowych platformach. Zbudowane testy stanowią aplikację automatycznie wykonującą przygotowane scenariusze testowe. Testy wykonano z użyciem Boost Unit Test Framework. Zastosowano także bibliotekę boost::random w celu dostarczenia losowych warunków wejściowych dla testowanych metod klas.

Narzędzia wykorzystane do wykonania projektu

Do wykonania projektu użyto narzędzi wymienionych poniżej:
•g++ / kompilator Microsoft Visual Studio 2010
•SCons
•gprof / profiler Microsoft Visual Studio 2010
•Mercurial, hosting na bitbucket.org
•Doxygen
•biblioteki Boost, w szczególności: Filesystem, Regex, Graph, Bind, Test
•szablony STL, w szczególności: vector, list, map

Diagram klas




Interface
Class
Fields
arguments_
es_
is_
os_
USAGE
Methods
argumentsError
directoryNotFoundError
filesystemError
getExtensions
getProjectDirectory
Interface
missingDependencyError
nonAcyclicGraph<CyclicEdgesVector>
nonConnectivityGraph
printUnnecessaryIncludes<Unnecessa...
Nested Types
FilesystemException : filesystem_error
Typedef
Project<SourceFile>
Template Class
Fields
files_
Methods
buildInclusionGraph
checkInclusionGraph
determineUnnecessaryIncludes
optimizeInclusionGraph
Project
Nested Types
Parser
Class
Fields
includes_
Methods
included
Parser
parseStreamForIncludes
readContent
Nested Types
InclusionGraph<Vertex>
Template Class
Fields
inclusionGraph_
isOptimized_
Methods
~InclusionGraph
addEdge
addVertex
depthFirstVisit
findOrder
findUnnecessaryIncludes
getUnnecessaryIncludes
InclusionGraph
isAcyclicGraph
isConnectivityGraph
join
Nested Types
DeptoolException
Class
exception
Fields
what_
Methods
~DeptoolException
DeptoolException (+ 1 overload)
what
Nested Types