Projektergebnis

Problemmusterkatalog mit Lösungsstrategien

Identifikation

Projektidentifikation	QBench
Projektergebnis	Problemmusterkatalog mit Lösungsstrategien
Vertraulichkeitsstufe	Öffentlich (ab 03/2006), vorher Verbundvertraulich
Arbeitspaket(e)	AP 5, AP6
Geplante Auslieferung	31.01.2006
Auslieferung	31.01.2006
Letzte Änderung	10.03.2006
Autor(en)	FIRST, FZI
Koordination	Thomas Mohaupt, FIRST

Zusammenfassung

Dieses Dokument enthält den Katalog der Problemmuster und der dazugehörigen Erkennungs- und Lösungsstrategien, der im Rahmen der Arbeitspakete 5 und 6 für die Identifizierung und Behebung von Qualitätsmängeln von objektorientierten Systemen erstellt wurde. Auf der Grundlage der hier aufgeführten Problem- und Lösungsmuster und mit Hilfe von entsprechenden Werkzeugen ist die Analyse, Bewertung und Verbesserung der inneren Qualität von Softwaresystemen möglich.

Schlüsselwörter

Problemmuster, Erkennungsstrategien, Lösungsmuster, Lösungsstrategien

Historie

Version	Datum	Autor(en)	Status & Änderungen
0.1	30.06.05	Mohaupt, Thomas	Draft
0.2	04.08.05	Mohaupt, Thomas	Einarbeitung Review
1.0	11.08.05	Mohaupt, Thomas	Aktualisierung einiger Problemmuster
1.1	20.01.06	Trifu, Adrian	Lösungsstrategien
1.2	10.03.06	Trifu, Adrian	PM „Lange Parameterliste“

Vertraulichkeitsstufe

	Klassifikation	Vertraulichkeitsstufe
X ab 03/06	Öffentlich	Der Öffentlichkeit zugängliche Informationen.
	Vertraulich	Den Mitgliedern des QBench-Verbundes und den Projektassoziierten vorbehaltene Informationen.
X bis 03/06	Verbundvertraulich	Den Mitgliedern des QBench-Verbundes vorbehaltene Informationen.

Haftungsausschluss

Die Autoren übernehmen keinerlei Gewähr für die Aktualität, Korrektheit, Vollständigkeit oder Qualität der in diesem Dokument bereitgestellten Informationen. Haftungsansprüche gegen die Autoren, welche sich auf Schäden materieller oder ideeller Art beziehen, die durch die Nutzung oder Nichtnutzung der dargebotenen Informationen bzw. durch die Nutzung fehlerhafter und unvollständiger Informationen verursacht wurden, sind grundsätzlich ausgeschlossen.

Inhaltsverzeichnis

1 Einleitung

Die Problemmuster im vorliegenden Katalog wurden von den QBench-Projektpartnern aus der Literatur oder basierend auf Erfahrungen, z.B. aus Softwareassessments, zusammengetragen.

1.1 Problemmusterklassifizierung

Während der Erfassung und Definition wurden die Problemmuster klassifiziert. Die Klassifizierung erfolgte hinsichtlich

- der Betrachtungsebene

Die Problemmuster wurden hierbei den Ebenen Technologie, Architektur, Design und/oder Code zugeordnet. Die Zuordnung leitet sich aus den vom jeweiligen Problemmuster betroffenen Artefakten ab. So werden z.B. Problemmuster, welche die Paketstruktur adressieren, der Architekturebene zugeordnet, während klassenspezifische Muster auf der Design- oder Codeebene angesiedelt sind.

- der Programmiersprache

Bei der Klassifizierung hinsichtlich der Programmiersprache wurde erfasst, inwieweit ein Problemmuster in den betrachteten Programmiersprachen Java, C++, C# und Delphi relevant ist bzw. ob es in diesen Sprachen auftreten kann.

- dem Grad der Automatisierung der Erkennung

Ein Ziel von QBench ist es, eine durchgängige Werkzeugunterstützung für die Verbesserung der Codequalität zu schaffen. Für den zugehörigen Prozess „Problemerkennung – Lösungsvorschlag – Problembehebung“ wird daher ein möglichst hoher Automatisierungsgrad angestrebt. Da die Problemerkennung und damit die Problemmuster am Anfang dieser Kette stehen, ist die Klassifizierung in Hinblick auf den möglichen Automatisierungsgrad der Erkennung von großer Bedeutung für den gesamten Prozess. Die Klassifizierung erfolgte letztlich in die zwei Kategorien: „Problemmuster mit automatischer Erkennung“ und „Problemmuster mit manueller Erkennung“ (s. u.).

Weitere Klassifizierungen fanden im Zusammenhang mit der Einbettung der Problemmuster in die Struktur der bidirektionalen Qualitätsmodelle statt. Dabei wurden die Problemmuster sowohl hinsichtlich ihrer Relevanz für verschiedene Qualitätseigenschaften als auch bezüglich der wirtschaftlichen Auswirkung einer möglichen Behebung der Probleme bewertet. Da diese Klassifizierung und Bewertungen inhaltlich dem Arbeitspaket 3 „Definition und Anwendung von Qualitätsmodellen, Metriken und Heuristiken“ zugeordnet werden, sind diese im nachfolgenden Katalog nicht ausgeführt.

1.2 Erkennungsstrategien

Neben der Sammlung der Problemmuster war die Entwicklung von zugehörigen Erkennungsstrategien ein gefordertes Ergebnis von AP5. Ausgehend von einer zunächst informalen Erkennungsstrategie wurde versucht, für jedes Problemmuster eine Erkennungsstrategie für das Analysewerkzeug SISSy [URLSISSy] in Form von SQL-Abfragen zu entwickeln. Dies war bis auf wenige Ausnahmen für alle Problemmuster möglich. Lediglich für Problemmustern, die auf Informationen beruhen, die nicht im Metamodell hinterlegt sind, konnten keine SQL-Abfragen erstellt werden, z.B. für das Problemmuster „duplizierter Code“. Einige dieser Problemmuster sind jedoch mit Hilfe zusätzlicher (zum Teil freiverfügbarer) Werkzeuge, z.B. mittels eines Duplikatcheckers, erkennbar.

1.3 Lösungsstrategien

Ziel des Arbeitspakets 6 war für die in AP5 zusammengestellten Problemmuster entsprechende, zunächst informelle Lösungsstrategien zu formulieren. Dabei wurde versucht, möglichst viele alternative Lösungswege

zu spezifizieren. Diese Lösungsstrategien sind als separater Abschnitt am Ende jedes Problemmusters hinzugefügt worden.

2 Problemmusterkatalog

Während der Präzisierung der Problemmusterdefinitionen wurden diese in die zwei Kategorien „automatisch erkennbare Problemmuster“ und „manuell erkennbare Problemmuster“ unterteilt. Problemmuster aus letzterer Kategorie sind dadurch geprägt, dass entweder

- kein Konsens über die Schwere oder die Lösung der Problemmuster erzielt werden konnte oder
- diese Problemmuster aufgrund ihrer Art grundsätzlich einer manuellen Anpassung bedürfen. Dies trifft zum Beispiel auf alle subsystemspezifischen Problemmuster zu, da die Zuordnung von Artefakten zu Subsystemen manuell erfolgen muss.

2.1 Problemmuster mit automatischer Erkennung

2.1.1 Allgemeine Parameter

Definition

Der Datentyp eines Methodenparameters wird innerhalb der Methode in einen spezielleren Typ umgewandelt (gecastet).

Beschreibung

Als Typ für einen Methodenparameter wird ein allgemeiner Datentyp angegeben. Innerhalb der Methode wird jedoch von einem spezielleren Datentyp des Parameters ausgegangen. Über einen Typecast wird der Parametertyp in diesen Datentyp umgewandelt und benutzt. Die Angabe des allgemeinen Datentyps suggeriert daher einem Klienten eine Allgemeinheit, die gar nicht vorhanden ist und von diesem somit auch nicht wie erwartet genutzt werden kann. Dies ist insbesondere deshalb gefährlich, weil Typfehler, die bei der Nutzung der vermuteten Allgemeinheit entstehen können, erst zur Laufzeit auftreten.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Innerhalb eines Methodenrumpfs werden alle Typumwandlungen (CAST) von Methodenparametern betrachtet.
- Eine Verletzung liegt für einen Parameter vor, sobald dieser mindestens einmal in einen spezielleren Typ umgewandelt wird.
- Jeder Parameter einer Methode wird separat betrachtet.

Erkennung mit SISSy

```
/* $Id: AllgemeineParameter.sql,v 1.2 2006/01/24 15:26:16 seng Exp $ */
/*
This Query looks for places in source code, where formal parameters are
downcasted.

*/

SELECT DISTINCT
        variables.name as Parameter,
        types.name as ParameterType,
        types2.name as CastedToType,
        functions.name as OccuringInFunction,
        types3.name as OccuringInClass,
        files.pathname as OccuringInFile
FROM
        TAccesses accesses1 join
        TVariables variables on (variables.id = accesses1.targetid) join
        TConstants cons on (variables.kindofvariable = cons.value and cons.name =
'VAR_FORMALPARAM') join
        TAccesses accesses2 on (accesses1.sourceid = accesses2.sourceid and
                accesses2.position = accesses1.position - 1) join
        /* Get the type of the formal parameter */
        TAccesses accesses3 on (accesses3.sourceid = variables.id) join
        /* Only downcasts are considered to be problematic */
        TInheritances inheritances on (accesses3.targetid = inheritances.superid and
inheritances.classid = accesses2.targetid) join
        /* In order to get the type */
        TTypes types on (accesses3.targetid = types.id) join
        /* Get the type that parameter was casted to */
        TTypes types2 on (accesses2.targetid = types2.id) join
        TFunctions functions on (accesses1.functionid = functions.id) join
        TTypes types3 on (functions.classid = types3.id) join
        TSourceEntities sourceEntities on (types3.id = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id)
;
```

Lösungsstrategie

Es muss untersucht werden, ob die Semantik der von der Methode bereitgestellten Funktionalität eventuell mit mehreren der Unterklassen des allgemeinen Typs sinnvoll kombinierbar ist. Wenn ja, dann bedeutet das, dass Teile der Methode in diese falsch platziert worden sind. In anderen Worten, die allgemeinen Teilen müssen von den speziellen Teilen getrennt werden, zum Beispiel durch die Entwurfsmuster „Strategy“ oder „Template method“ Entwurfsmuster. Eventuell können die Untertyp-spezifischen Teile direkt in den speziellen Typ, als spezielle Implementierung einer im Obertyp definierten abstrakten Methode, verschoben werden.
Wenn die Methode Funktionalität enthält die nur für den Untertyp bestimmt ist, kann die Methodensignatur so geändert werden dass der Untertyp übergeben wird.

2.1.2 Attributüberdeckung

Definition

Der Name eines nicht-statischen Attributes einer Oberklasse wird in mindestens einer direkten oder indirekten Unterklasse für ein anderes Attribut erneut verwendet. Hierbei ist die Sichtbarkeit und der Typ der betrachteten Attribute irrelevant und eine Übereinstimmung nicht erforderlich.

Beschreibung

Eine Unterklasse deklariert ein Attribut, das mit dem gleichen Namen bereits in einer direkten oder indirekten Oberklasse vorkommt. Wird dieses Attribut über seinen Namen angesprochen, verdeckt es das gleichnamige Attribut in der Oberklasse (wenn das Attribut in der Unterklasse sichtbar wäre) oder verwendet lediglich denselben Namen für eine andere Variable. Auch wenn die Sprachen Java und C++ hierin kein Problem sehen und jeweils separate Speicherbereiche zuordnen, so bedeutet die namentliche Mehrdeutigkeit für die Wartung große Risiken und Zusatzaufwände.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Gesucht werden alle nicht-statischen Attribute für die jeweils gilt: Der Name des Attributs wird in mindestens einer direkten oder indirekten Unterklasse (der Klasse des Attributs) erneut für ein Attribut verwendet.
- Jedes derartige Oberklassenattribut stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
create temp table VHiddenAttributes as


SELECT
       meSuperClass.id AS superclassid
       , meSuperClass.fullname AS superclass
       , membSuperAttr.name AS superclassAttribute
       , membSuperAttr.id AS superAttributeID
       , COUNT (membSubAttr.id) AS numberOfOverwrites

FROM
       TTypes meSubClass join
       TInheritances inh on (inh.classId = meSubClass.id) join
       TTypes meSuperClass on (inh.superId = meSuperClass.id) join
       TMembers membSubAttr on (membSubAttr.classId = meSubClass.id) join
       TMembers    membSuperAttr    on    (membSuperAttr.classId    =    meSuperClass.id    AND
membSubAttr.name = membSuperAttr.name) join
       TConstants tcAttr on ( membSuperAttr.kindOfMember       =       tcAttr.value       AND
membSubAttr.kindOfMember = tcAttr.value)

WHERE
       tcAttr.name = 'VAR_FIELD'

       AND membSuperAttr.isstatic = 0
       AND membSuperAttr.name <> '<self>'

GROUP BY
       membSuperAttr.id
       , meSuperClass.id
       , meSuperClass.fullname
       , membSuperAttr.name

;


/* Nachbearbeitung: Beteiligte Dateien ausgeben */
SELECT
       hiddenAttributes.superclass,
       hiddenAttributes.superclassAttribute,
       files2.pathname,
       hiddenAttributes.numberOfOverwrites

FROM
       VHiddenAttributes as hiddenAttributes join
       TSourceEntities      sourceEntities2      on      (hiddenAttributes.superclassid      =
sourceEntities2.id) join
       TFiles files2 on (sourceEntities2.sourcefileid = files2.id)

;

drop table VHiddenAttributes;
```

Lösungsstrategie

Bei typgleichen Attributen ist unbedingt zu prüfen, ob das Attribut der Unterklasse entfernt, und stattdessen das Attribut der Oberklasse benutzt werden kann. Wenn aber keine semantische Übereinstimmung zwischen den beiden Attributen besteht ist es trotzdem empfehlenswert den Attributen unterschiedliche Namen zu geben, um eventuellen Missverständnisse während Wartungsaktivitäten zu vermeiden.

2.1.3 Ausgeschlagenes Erbe (Implementierung)

Definition

Eine Methodenimplementierung einer Klasse OK wird in der Mehrheit der direkten Unterklassen von OK nicht-leer redefiniert ohne das geerbte Pendant aus OK zu verwenden, d.h. die geerbte Implementierung wird von den Unterklassen ausgeschlagen.

Beschreibung

Im Allgemeinen, dürfen Unterklassen geerbte Methoden, die zur Schnittstelle der Oberklasse gehören, mit spezialisierten Versionen überschreiben. Wenn dies aber in der Mehrheit der Unterklassen passiert, ist das ein Zeichen dafür, dass die geerbte Implementierung nicht allgemein genug ist. Dies gilt insbesondere, wenn die Re-Implementierungen ohne die Verwendung der geerbten Methode auskommen (d.h. diese nicht aufrufen), da in diesen Fällen kein inhaltlicher Zusammenhang zwischen den beiden Methoden ersichtlich ist. In diesen Fällen ist die gewählte Vererbungsstruktur offensichtlich nicht optimal.

Programmiersprache

C++, Java, C# , Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermittelt werden alle implementierten Methoden in Oberklassen für die gilt: Die Methode wird in der Mehrheit (>50%) der direkten Unterklassen nicht-leer überdeckt, ohne dabei aufgerufen zu werden.
- Jede derartige Oberklassenmethode stellt eine Probleminstanz dar

Erkennung mit SISSy

```
/* $Id: AusgeschlagenesErbeImplementierung.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */



CREATE Table VMethodOverrides AS

SELECT
        A.id as SubClassId,
        A.fullname as SubClassFullName,
        FA.id as OverrideMethodId,
        FA.name as OverrideMethodName,
        B.id as SuperClassId,
        B.fullname as SuperClassFullName,
        FB.id as OverridenMethodId,
        FB.name as OverridenMethodName
FROM
        TTypes A,
        TTypes B,
        TFunctions FA,
        TFunctions FB,
        TMembers A_Am,
        TMembers B_Bm,
        TConstants visibility
WHERE
        A.id = A_Am.classID AND
```

```
        FA.id = A_Am.id AND
        B.id = B_Bm.classID AND
        FB.id = B_Bm.id AND
        A_Am.overridenmemberid = B_Bm.id AND

        /* Die Methoden dürfen nicht "leer" sein */
        FA.NumberOfStatements > 0 AND
        FB.NumberOfStatements > 0 AND


        /* Die Methode ist public/protected */
        B_Bm.visibility = visibility.value AND
        (
                (visibility.name = 'VISIBILITY_PUBLIC' OR
                        visibility.name = 'VISIBILITY_PROTECTED') OR

                /* Oder die Methode ist package-local deklariert und A ist in dem selben Paket
wie parent */
                (visibility.name = 'VISIBILITY_PACKAGE' AND
                        A.packageId = B.packageId)

        )/*AND*/


EXCEPT

SELECT
        A.id as SubClassId,
        A.fullname as SubClassFullName,
        FA.id as OverrideMethodId,
        FA.name as OverrideMethodName,
        B.id as SuperClassId,
        B.fullname as SuperClassFullName,
        FB.id as OverridenMethodId,
        FB.name as OverridenMethodName
FROM
        TTypes A,
        TTypes B,
        TFunctions FA,
        TFunctions FB,
        TMembers A_Am,
        TMembers B_Bm,
        TConstants visibility,
        TAccesses accesses
WHERE
        A.id = A_Am.classID AND
        FA.id = A_Am.id AND
        B.id = B_Bm.classID AND
        FB.id = B_Bm.id AND
        A_Am.overridenmemberid = B_Bm.id AND

        /* Die Method A_m ist fast leer
        FA.NumberOfStatements > 0 AND
        FB.NumberOfStatements > 0 AND
                */

        /* Die Methode ist public/protected */
        B_Bm.visibility = visibility.value AND
        (
                (visibility.name = 'VISIBILITY_PUBLIC' OR
                        visibility.name = 'VISIBILITY_PROTECTED') OR

                /* Oder die Methode ist package-local deklariert und A ist in dem selben Paket
wie parent */
                (visibility.name = 'VISIBILITY_PACKAGE' AND
                        A.packageId = B.packageId)

        )/*AND*/ AND

        /* Die Methode der Unterklasse ruft die Methode der Oberklasse auf, ergänzt sie
sozusagen */
        accesses.targetid = FB.id AND
        accesses.functionid = FA.id
;


CREATE Table VMethodsPerClass AS

/* Wie viele Methoden haben die Klassen, die vererbt werden, Konstruktoren und Desktruktoren
werden nicht mitgezählt */
```

```
SELECT
                type.id as TypeId,
                type.fullName as TypeName,
                count(distinct func.id) as numberOfMethods
FROM
                TTypes as type join
                TModelElements as elem on (type.id = elem.parentId) join
                TFunctions as func on (elem.parentid = func.classId) join
                TConstants as con on (elem.kindofelement = con.value) join
                TMembers as member on (member.id = func.id) join
                TConstants as constants on (member.visibility = constants.value)
WHERE
                elem.id = func.id and
                (con.name like 'FUNC_METHOD') AND
                member.isstatic = 0 AND
                member.isfinal = 0  AND
                constants.name != 'VISIBILITY_PRIVATE'
GROUP BY
                type.id,
                type.fullName

;


CREATE Table VMethodsInherited AS


SELECT
        type.id as TypeId,
        type.fullName as TypeName,
        SUM (mpc.numberOfMethods) as numberOfInheritedMethods
FROM
        TTypes as type join
        TInheritances as inheritance on (type.id = inheritance.classid) join
        VMethodsPerClass as mpc on (mpc.TypeId = inheritance.superid)
GROUP BY
        type.id,
        type.fullName

;

CREATE Table VOverridesPerClass AS

/* Wie viele Methoden der Klasse überschreiben eine Methode der Oberklasse */

SELECT
                type.id as TypeId,
                type.fullName as TypeName,
                count(distinct methodOverrides.OverrideMethodId) as numberOfOverrides
FROM
                TTypes as type join
                VMethodOverrides as methodOverrides on (type.id = methodOverrides.SubClassId)
GROUP BY
                type.id,
                type.fullName


;

/*

Problemmuster geändert, nur noch Teil 2 der Erkennung relevant
Dieser Teil ist nun Generationskonflikt

SELECT
        mi.TypeName as ClassName,
        mi.numberOfInheritedMethods * 1.0 as NumberOfInheritedMethods,
        opc.numberOfOverrides * 1.0 as NumberOfOverrides,
        (opc.numberOfOverrides*1.0) / (mi.numberOfInheritedMethods*1.0) as Ratio,
        files.pathname

FROM
        VMethodsInherited as mi join
        VOverridesPerClass as opc on (mi.TypeId = opc.TypeId) join
        TSourceEntities se on (mi.TypeId = se.id) join
        TFiles files on (se.sourcefileid = files.id)


WHERE
```

```
        (opc.numberOfOverrides*1.0) / (mi.numberOfInheritedMethods*1.0) > 0.66


;

*/

/* Teil 2: Eine Methode wird in mehr als 66% der direkten und indirekten Unterklassen
überschrieben */


CREATE Table VNumberOfSubClasses as

SELECT
        ti.superid,
        count(distinct ti.classid) as numberOfSubclasses

FROM
        TInheritances as ti
GROUP BY
        ti.superid

;

CREATE Table VNumberOfOverrides AS

SELECT
        mo.OverridenMethodID,
        mo.OverridenMethodName,
        mo.SuperClassId,
        mo.SuperClassFullName,
        count(DISTINCT mo.OverrideMethodId) as NumberOfOverrides
FROM
        VMethodOverrides as mo
GROUP BY
        mo.OverridenMethodID,
        mo.OverridenMethodName,
        mo.SuperClassId,
        mo.SuperClassFullName
;


/* The number of subclasses indicates the maximum number of
potential overrides, so that's our threshold */

SELECT
        noo.OverridenMethodName,
        noo.SuperClassFullName,
        noo.NumberOfOverrides,
        nosc.NumberOfSubClasses,
        (noo.NumberOfOverrides*1.0) / (nosc.NumberOfSubClasses*1.0),
        files.pathname

FROM
        VNumberOfOverrides as noo join
        VNumberOfSubClasses as nosc on (noo.superClassId = nosc.superid) join
        TSourceEntities se on (noo.SuperClassId = se.id) join
        TFiles files on (se.sourcefileid = files.id)

WHERE
        (noo.NumberOfOverrides*1.0) / (nosc.NumberOfSubClasses*1.0) > 0.50
;


DROP Table VMethodsInherited;
DROP Table VNumberOfOverrides;
DROP Table VNumberOfSubClasses;
DROP Table VOverridesPerClass;
DROP Table VMethodsPerClass;
DROP Table VMethodOverrides;
```

Lösungsstrategie

Im Allgemeinen ist dieses Problem ein Hinweis, dass die überschriebene Methode in der Oberklasse „zu hoch“ in der Vererbungshierarchie platziert ist. In diesem Fall muss geklärt werden, ob die Verwendung des Strategie-Entwurfsmusters angemessen ist. Wenn ja, kann diese folgendermaßen implementiert werden:

1. Es wird eine Strategie-Schnittstelle definiert

2. Die Implementierung der Methode der Oberklasse wird in eine Unterklasse der Strategie-Schnittstelle ausgelagert.
3. In der Oberklasse wird die Implementierung durch einen Aufruf der Strategie ersetzt
4. Die (überschriebenen) Implementierungen in den Unterklassen werden ebenfalls in Strategieklassen ausgelagert

Alternativ, wenn das Strategie-Entwurfmuster nicht sinnvoll ist, besteht die allgemeine Lösung in der Verschiebung der überschriebenen Methode „nach unten“ in der Vererbungshierarchie. Konkret, werden folgende Schritte durchgeführt:

1. Es wird eine neue Unterklasse (UkNeu) deklariert, in die die Implementierung der überschriebenen Methode aus der Oberklasse verschoben wird
2. Alle Unterklassen, die die verschobene Methode nicht überschreiben, werden im Vererbungsbaum unter die neue Unterklasse (UkNeu) verschoben, erben also jetzt statt von der Oberklasse von der neuen Unterklasse
3. Die Oberklasse wird in eine abstrakte Klasse (oder sofern möglich in ein Interface) umgewandelt.
4. Die Implementierung der überschriebenen Methode wird in der Oberklasse gelöscht. -> abstrakte Methode bzw. Interface-Methode.
5. Instanziierungen der Oberklasse werden durch Instanziierungen der neuen Unterklasse (UkNeu) ersetzt

2.1.4 Ausgeschlagenes Erbe (Schnittstelle)

Definition

Eine Unterklasse unterstützt nicht die vollständige Schnittstelle ihrer Oberklasse bzw. ihres Interfaces, d.h. dass Sie entweder mindestens

- eine Methode der Schnittstelle der Oberklasse (d.h. Deklaration oder Implementierung) durch eine leere Methode überschreibt oder implementiert (inkl. des je nach Rückgabewertes minimal notwendigen "return null"), oder
- die Sichtbarkeit einer Methode der Schnittstelle der Oberklasse reduziert (public>protected>private) (nur C++).

Beschreibung

Im Allgemeinen wird bei der Verwendung von Vererbung von der Gültigkeit des *Liskov Substitution Principle (LSP)* ausgegangen. Dieses besagt, dass Objekte einer abgeleiteten Klasse grundsätzlich anstelle eines Basisklassenobjekts verwendet werden können. Die Voraussetzung dafür ist die vollständige Unterstützung aller geerbten Schnittstellen durch die Unterklasse.
Eine Klasse verstößt gegen das LSP, wenn sie die Schnittstelle ihrer Oberklasse nicht vollständig unterstützt. Das ist dann der Fall, wenn die Klasse Methoden der Oberklassen mit leeren Methoden überschreibt bzw. implementiert oder deren Sichtbarkeit einschränkt (im letzteren Fall wird die Schnittstelle nur noch "heimlich" über den Typ der Oberklasse unterstützt; durch die Unterklasse ist sie nicht mehr vorhanden).
Die Verwendung solcher Klassen als Instanzen der Oberklasse kann zu unerwarteten Laufzeitproblemen führen, denn Klienten verwenden normalerweise die Schnittstelle der Oberklasse und erwarten, dass alle Unterklassenobjekte diese Schnittstelle auch vollständig unterstützen.
Referenzen: [Fowler 99] [Riel 96]

Programmiersprache

C++, Java

Betrachtungsebene

Designebene

Informale Erkennung

Ermittelt werden alle Methoden von Unterklassen, die eine Methode ihrer Oberklasse bzw. Schnittstelle entweder

- mit einer Methode mit kleinerer Sichtbarkeit überschreiben (public>protected>private; nur C++), oder

- mit einer leeren Implementierung (bzw. mit dem evtl. minimal notwendigen return null) überschreiben oder implementieren.

Jedes Methodenpaar (Unterklassenmethode, Oberklassenmethode), das diese Forderungen erfüllt, stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: AusgeschlagenesErbeSchnittstelle.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
  Beschreibung:
  ------------

       Der Mechanismus des Überschreibens von Methodenimplementierungen ist primär dazu
geeignet, polymorphe Aufrufstrukturen zu ermöglichen.
       Grundsätzlich sollte das Liskov-Substitution-Principle gelten: Ein Objekt einer
abgeleiteten Klasse sollte grundsätzlich an die Stelle
       eines Objekts der Basisklasse treten können.

       Eine Verletzung dieses Prinzips liegt nahe, wenn eine der folgenden Situationen
vorgefunden wird:

              *  Eine Methodenimplementierung der Basisklasse wird durch eine leere
Methodenimplementierung in der abgeleiteten Klasse
                ersetzt (dann wurde die Methode ggf. zu früh in der Vererbungshierarchie
eingeführt).
              * Die Sichtbarkeit einer Methode wird in der abgeleiteten Klasse eingeschränkt
(möglich in C++).
              *  Die Methodenimplementierung der abgeleiteten Klasse überschreibt die
Implementierung der Basisklasse völlig (eher ein weiches Kriterium, weil
                dies manchmal tatsächlich erwünscht wird)
              *  Die Methodenimplementierung der abgeleiteten Klasse tut inhaltlich etwas
völlig anderes (erkennbar ggf. an Nichtnutzung von Parametern)


   Erkennung des Problemmusters:
   ----------------------------

       Analyse der Methodenimplementierungen entlang der Vererbungshierarchie und ggf. der
Call-Beziehungen die Hierachie "hinauf" gerichtet.
       Inhaltliche Analyse über den Zweck der jeweiligen Implementierungen nur über unscharfe
heuristiken möglich (z.B. nutzen ähnliche Klassen, Nutzung der Parameter)

*/


/* Eine Methodenimplementierung der Basisklasse wird durch eine leere Methodenimplementierung
in der abgeleiteten Klasse
                ersetzt (dann wurde die Methode ggf. zu früh in der Vererbungshierarchie
eingeführt).
*/
/* @TODO:
Überschreibende Methode löst nur eine Ausnahme aus
(SISSY muss noch erweitert werden für performante Implementierung
 */

SELECT DISTINCT
       A.id as SubClassId,
       A.fullname as SubClassFullName,
       FA.id as OverrideMethodId,
       FA.name as OverrideMethodName,
       B.id as SuperClassId,
       B.fullname as SuperClassFullName,
       FB.id as OverridenMethodId,
       FB.name as OverridenMethodName,
       returnAccesses2.targetid,
       returnAccesses.targetid,
       files.pathname as filename_A,
       files2.pathname as filename_B
FROM
       TFunctions FA join
       TMembers A_Am on (A_Am.id = FA.id) join
       TTypes A on (A.id = A_Am.classId) join
       TMembers B_Bm on (B_Bm.id = A_Am.overridenmemberid) join
       TFunctions FB on (B_Bm.id = FB.id) join
       TTypes B on (B.id = B_Bm.classid) join
       TInheritances AB on (A.id = AB.classid and AB.superID = B.id) join
```

```
        TConstants visibility on (B_Bm.visibility = visibility.value) join
        TAccesses returnAccesses on (fb.returntypedeclarationid = returnAccesses.id) join
        TAccesses returnAccesses2 on (fa.returntypedeclarationid = returnAccesses2.id) join
        TStatements aStatements on (aStatements.functionid = FA.id) join
        TConstants statementtype on (aStatements.kindofstatement = statementtype.value) join
        TSourceEntities sourceEntities on (A.id = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id) join
        TSourceEntities sourceEntities2 on (B.id = sourceEntities2.id) join
        TFiles files2 on (sourceEntities2.sourcefileid = files2.id)

WHERE


        /* Die Methode ist public/protected */
        B_Bm.visibility = visibility.value AND
        (
                (visibility.name = 'VISIBILITY_PUBLIC' OR
                        visibility.name = 'VISIBILITY_PROTECTED') OR

                /* Oder die Methode ist package-local deklariert und A ist in dem selben Paket
wie parent */
                (visibility.name = 'VISIBILITY_PACKAGE' AND
                        A.packageId = B.packageId)

        )

        AND
                (
                (       /* Die Method A_m ist leer */
                FA.NumberOfStatements = 0 AND
                FB.NumberOfStatements > 0
                )


                /* Die überschreibende Methode enthält als einziges Statement ein return null,
die überschriebene nicht */

                OR

                (FA.NumberOfStatements = 1 AND
                        statementtype.name = 'STATEMENT_RETURN' AND
                        NOT EXISTS (
                                SELECT
                                        *
                                FROM
                                        TStatements statements join
                                        TAccesses     accesses     on     (statements.id     =
accesses.sourceid) join
                                        TConstants   cons   on   (statements.kindofstatement   =
cons.value)
                                WHERE
                                        statements.functionid = FA.id AND
                                        cons.name = 'STATEMENT_RETURN'
                        )
                )

                /*  Die  überschreibende  Methode  enthält  als  einziges  Statement  ein  throw
Statement */


                OR

                (FA.NumberOfStatements = 1 AND
                statementtype.name = 'STATEMENT_THROW'
                )

)


;
```

Lösungsstrategie

Da dieses Problem ein Hinweis auf eine falsche Vererbungsstruktur ist, kann keine allgemeine Lösung angegeben werden. Im Allgemeinen muss man sicherstellen dass die Semantik aller existierenden Vererbungsbeziehungen gewährleistet. Dazu kann man die Heuristiken aus [Riel96] verwenden.

Die Vererbungsbeziehung muss überdacht werden, und eventuell durch Komposition ersetzt. Eine alternative Möglichkeit ist, die "ausschlagende" Unterklasse im Vererbungsbaum zu verschieben, oder es kann versucht werden, eine gemeinsame Schnittstelle der Oberklasse und der Unterklasse zu finden, die auf der einen Seite so groß wie möglich ist, andererseits aber auch von beiden Klassen erfüllt wird.

2.1.5 Datenkapselaufbruch

Definition

Eine nicht-geschachtelte Klasse bietet Attribute (keine Konstanten) öffentlich an, auf die von äußeren Klassen zugegriffen wird (lesend oder schreibend).

Beschreibung

Ein Grundziel des objektorientierten Vorgehens ist die Datenkapselung, d.h. die objektorientierten Klassen fungieren als abstrakte Datentypen, die lediglich Services anbieten, die ihrerseits auf klasseninternen Daten basieren. Durch diese Form der Datenkapselung ist die Klasse in der Lage, die Korrektheit und Gültigkeit der internen Daten sicherzustellen, indem die Services entsprechende Überprüfungsmechanismen realisieren und Abhängigkeiten berücksichtigen.
Anderenfalls können ungültige Objekte existieren, deren Weiterverarbeitung im System zu Folgefehlern führen kann, deren Ursprung dann nur mit großen Zusatzaufwänden aufzufinden ist. Die geschickte Datenkapselung führt daher die Gültigkeitsprüfung bei jedem Ändern einer der Daten durch. Damit dies möglich ist, dürfen direkte Verwendungen der Daten nicht mehr möglich sein.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermittlung aller öffentlichen, nicht-konstanten Attribute von nicht-geschachtelten Klassen.
- Betrachtet werden nur die Attribute, die von einer äußeren Klasse lesend oder schreibend verwendet werden.
- Jedes Attribut auf das diese Forderungen zutreffen, stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: DatenkapselAufbruch.sql,v 1.2 2006/01/19 13:51:17 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*Datenkapselaufbruch

Beschreibung:
 Eine Klasse bietet Atribute öffentlich an, die auch von aussen direkt gelesen oder/und
geschrieben werden. Dies stellt eine grobe Verletzung des OO-Gedankens der Datenkapselung dar.

Implementierung:
  - alle öffentliche Attribute suchen ..
  - auf die (lesend/schreibend) ..
  - aber nicht von der selben Klasse oder von einer ihrer Unterklassen zugegriffen wird
  - und keine Konstante (static final) ist
*/

CREATE OR REPLACE VIEW VEncapsulationViolations AS

SELECT DISTINCT
       classOfMethod.id AS ContainingClassIdOfAccessingFunction,
       classOfMethod.fullname AS ContainingClassNameOfAccessingFunction,
       method.id as AccessingFunctionId,
       method.name as AccessingFunctionName,
```

```
        classOfField.id AS ContainingClassIdOfField,
        classOfField.fullName as ContainingClassNameOfField,
        field.id AS AccessedFieldId,
        field.name as AccessedFieldName
FROM
        TTypes classOfField,
        TMembers member_classField,
        TVariables field,
        TConstants constantField,

        TTypes classOfMethod,
        TMembers member_classMethod,
        TFunctions method,
        TConstants constantPublic,

        TAccesses tAccesses
WHERE
        /* field */
        member_classField.ClassId = classOfField.Id AND
        field.Id = member_classField.Id AND
        field.KindOfVariable = constantField.Value AND
        constantField.Name = 'VAR_FIELD' AND

        member_classField.visibility = constantPublic.Value AND
        constantPublic.Name = 'VISIBILITY_PUBLIC' AND
        NOT (member_classField.isStatic = 1 AND
             member_classField.isFinal = 1) AND

        /* method */
        member_classMethod.classId = classOfMethod.id AND
        method.id = member_classMethod.id AND

        /* method and field of different classes*/
        classOfField.id != classOfMethod.id AND

        /* method access field*/
        tAccesses.targetId = field.id AND
        method.id = tAccesses.functionId AND

        /* field class MUST NOT be an inner class */
        classOfField.classid = -1 AND

        /* method class MUST NOT be a sub class of field class*/
        NOT EXISTS (
                SELECT
                        i.classid
                FROM
                        TInheritances i
                WHERE
                        i.classID = classOfMethod.id AND
                        i.superID = classOfField.id

                UNION

                SELECT
                        tcr.classid
                FROM
                        TClassContainmentRelations tcr
                WHERE
                                classOfMethod.id = tcr.classid
                        AND     classOfField.id = tcr.containingclassid
        )
;


/* Nachbearbeitung: Beteiligte Dateien ausgeben */
SELECT DISTINCT
        files2.pathname,
        violations.AccessedFieldName,
        violations.AccessedFieldId

FROM
        VEncapsulationViolations as violations join
        TSourceEntities sourceEntities on (violations.ContainingClassIdOfAccessingFunction =
sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id) join
        TSourceEntities    sourceEntities2    on    (violations.ContainingClassIdOfField    =
sourceEntities2.id) join
        TFiles files2 on (sourceEntities2.sourcefileid = files2.id)
GROUP BY
```

```
        violations.AccessedFieldId,
        violations.AccessedFieldName,
        files2.pathname
;

drop view VEncapsulationViolations;
```

Lösungsstrategie

Als erstes muss überlegt werden ob das Attribut überhaupt in die Klasse gehört oder ob es sich um ein deplaziertes Attribut handelt, das in eine der Klassen gehört, die am meisten auf das Attribut zugreifen. Nachdem das Attribut korrekt platziert wurde, muss eine Sichtbarkeit (am besten private-Sichtbarkeit) für das Attribut ausgewählt werden. Ziel soll sein, das Attribut hinter höherwertigen Services zu „verstecken". Die einfache Ersetzung der Direktzugriffe durch Getter- und Setter-Methoden ist zwar besser als nichts, aber in der Regel keine vollständige Lösung.
Eine vollständige Lösung des Problems kann nur als Resultat einer Analyse der Schnittstelle der Klasse selbst und der Klienten des Attributs erfolgen, gefolgt von einer Restrukturierung der Klasse und der Klienten, so dass zumindest Zustandsänderungen in der Klasse möglichst durch höherwertige Services und nicht direkt über setter Methoden erfolgt.

2.1.6 Duplizierter Code

Definition

Duplizierter Code liegt vor, wenn mindestens 40 aufeinander folgende Codezeilen (inkl. Leerzeilen und Kommentarzeilen) innerhalb des Quellcodes in identischer Form mehrmals auftreten.

Beschreibung

Gleiche oder ähnliche Funktionalität muss häufig an mehreren Stellen genutzt werden. Anstatt jedoch die Funktionalität heraus zu faktorisieren und so nutzbar zu machen, wird sie unter Verwendung von Copy und Paste (Kopieren und Einfügen) dupliziert. So können Duplikate entstehen, die Methodenteile, ganze und mehrere Methoden, komplette Klassen oder auch vollständige Dateien umfassen. Das mehrfache Auftreten identischer Code-Fragemente bedeutet mehrfache Aufwände beim Verstehen, Warten und Testen. Insbesondere Änderungen müssen konsistent mehrfach durchgeführt werden, was wiederum sehr fehleranfällig ist.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Technologieebene

Informale Erkennung

- Ermittelt werden alle Code-Duplikate, bei denen mindestens 40 Zeilen identisch auftreten.
- Gezählt wird die Anzahl der von Code-Duplikaten betroffenen Zeilen. Dieser Wert ergibt sich, indem die einzelnen Fragmente der Duplikate bzgl. ihrer Länge summiert werden, wobei jedoch jede Codezeile maximal einmal gezählt wird. Diese Zählung entspricht folgendem Vorgehen: Der gesamte Quellcode wird ausgedruckt und alle Code-Duplikate markiert. Gezählt werden dann alle markierten Zeilen.

Erkennung mit SISSy

Erkennung mit Duplikat-Checker, z.B. CloneAnalyzer ([URLCloneAnalyzer])

Lösungsstrategie

Duplizierter Code kann verschiedene Ursachen und viele Erscheinungsformen haben. Eine der häufigsten und einfachsten Formen ist, wenn dasselbe Codefragment in zwei oder mehreren Methoden derselben Klasse auftaucht. In diesem Fall empfiehlt sich diese Fragmente als neue Methode der Klasse zu extrahieren und sie durch Methodenaufrufe zu ersetzten.
Wenn Codeduplikation zwischen Geschwisterklassen besteht, können die extrahierten Methoden „nach oben“ in der Vererbungshierarchie verschoben werden. Wenn es leichte Unterschiede zwischen den Unterklassenspezifischen Versionen gibt kann man das Entwurfsmuster „template method“ verwenden, um die gemeinsamen Teile in die Oberklasse zu ziehen.
Schließlich, in dem Fall dass Codeduplikation zwischen unverwandten Klassen besteht, gibt es generell zwei Möglichkeiten: entweder gehört die Funktionalität in einer der Klassen und die andere soll die entsprechende Methode aufrufen, oder die Funktionalität gehört in keiner der zwei Klassen. In diesem Fall empfiehlt es sich eine neue Klasse zu erzeugen, die die gemeinsame Funktion enthält und welche von den ursprünglichen Klassen referenziert wird.

2.1.7 Falsche Namenslänge

Definition

Die Länge des Namens eines Artefakts ist kleiner als 2 oder größer als 50. Folgende Artefakte werden hierbei betrachtet:

- Pakete (jeweils ohne Pfad)
- Dateien (jeweils ohne Extension)
- Klassen/Interfaces
- Methoden (ohne Parameter und Rückgabetyp)
- Attribute/Konstanten

Beschreibung

Bezeichner von Artefakten auf der einen Seite ausdrucksstark sein und Rückschlüsse auf ihren Zweck ermöglichen, auf der anderen Seite aber noch lesbar und einprägsam sein. So gewählte Artefaktnamen erhöhen die Lesbarkeit des Codes und damit dessen Wartbarkeit. Kontraproduktiv ist es,

- wenn die Länge des Artefaktnamens zu groß wird, d.h. nur mit erhöhtem Aufwand lesbar und wahrnehmbar ist, und
- wenn die Länge des Artefaktnamens =1 ist, d.h. kaum Inhalte vermittelt und eine große Gefahr besteht, gleichbenannte Artefakte im System zu besitzen (die dann u.U. z.B. eine unbewusste Attributüberdeckung bewirken).

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

- Ermitteln der Namenslänge der relevanten Artefakte (s.o.).
- Jedes Artefakt, dessen Länge kleiner als 2 oder größer als 50 ist, stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/*
 * $Id: DuplizierterCode.sql,v 1.1 2006/01/18 15:25:31 seng Exp $
 */
```

Lösungsstrategie

Die Lösung besteht im Umbenennen der Artefakte, durch prägnante Namen. Hier müssen eventuelle Namenskonventionen berücksichtigt werden (z.B. ungarische Notation).

2.1.8 Generationskonflikt

Definition

Sehr wenige, oder eine direkte Unterklasse UK einer Oberklasse OK reimplementiert nicht-leer mehr als 50% der in der Oberklasse implementierten und in die Unterklasse geerbten Methoden. Die Reimplementierung verwendet hierbei jeweils nicht das geerbte Methoden-Pendant.

Beschreibung

Die bereits schwierige Form der Vererbung für Source-Code-Sharing wird beim Generationskonflikt weiter verschärft, in dem ein Großteil der geerbten Methoden vollständig neu implementiert wird. Die semantische Ähnlichkeit der Methoden kann somit in keinster Weise mehr sichergestellt werden. Verschärt wird diese Problem noch dadurch, dass hier nur die Fälle betrachtet werden, in denen die Oberklassenmethode nicht verwendet bzw. aufgerufen wird. Eine Typhierarchie kann damit i.A. nicht mehr umgesetzt werden.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermittlung aller Unterklassen für die gilt: Die Unterklasse überschreibt nicht-leer mehr als 50% der in einer direkten Oberklasse implementierten und in der Unterklasse sichtbaren Methoden, ohne die Oberklassenmethode aufzurufen.

- Jede ermittelte Unterklasse-Oberklasse Paar stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* FZI Forschungszentrum Informatik */


CREATE Table VMethodOverrides AS

SELECT
        A.id as SubClassId,
        A.fullname as SubClassFullName,
        FA.id as OverrideMethodId,
        FA.name as OverrideMethodName,
        B.id as SuperClassId,
        B.fullname as SuperClassFullName,
        FB.id as OverridenMethodId,
        FB.name as OverridenMethodName
FROM
        TTypes A,
        TTypes B,
        TFunctions FA,
        TFunctions FB,
        TMembers A_Am,
        TMembers B_Bm,
        TConstants visibility
WHERE
        A.id = A_Am.classID AND
```

```
        FA.id = A_Am.id AND
        B.id = B_Bm.classID AND
        FB.id = B_Bm.id AND
        A_Am.overridenmemberid = B_Bm.id AND

        /* Die Methoden dürfen nicht "leer" sein */
        FA.NumberOfStatements > 0 AND
        FB.NumberOfStatements > 0 AND


        /* Die Methode ist public/protected */
        B_Bm.visibility = visibility.value AND
        (
                (visibility.name = 'VISIBILITY_PUBLIC' OR
                        visibility.name = 'VISIBILITY_PROTECTED') OR

                /* Oder die Methode ist package-local deklariert und A ist in dem selben Paket
wie parent */
                (visibility.name = 'VISIBILITY_PACKAGE' AND
                        A.packageId = B.packageId)

        )/*AND*/


EXCEPT

SELECT
        A.id as SubClassId,
        A.fullname as SubClassFullName,
        FA.id as OverrideMethodId,
        FA.name as OverrideMethodName,
        B.id as SuperClassId,
        B.fullname as SuperClassFullName,
        FB.id as OverridenMethodId,
        FB.name as OverridenMethodName
FROM
        TTypes A,
        TTypes B,
        TFunctions FA,
        TFunctions FB,
        TMembers A_Am,
        TMembers B_Bm,
        TConstants visibility,
        TAccesses accesses
WHERE
        A.id = A_Am.classID AND
        FA.id = A_Am.id AND
        B.id = B_Bm.classID AND
        FB.id = B_Bm.id AND
        A_Am.overridenmemberid = B_Bm.id AND

        /* Die Method A_m ist fast leer
        FA.NumberOfStatements > 0 AND
        FB.NumberOfStatements > 0 AND
                */

        /* Die Methode ist public/protected */
        B_Bm.visibility = visibility.value AND
        (
                (visibility.name = 'VISIBILITY_PUBLIC' OR
                        visibility.name = 'VISIBILITY_PROTECTED') OR

                /* Oder die Methode ist package-local deklariert und A ist in dem selben Paket
wie parent */
                (visibility.name = 'VISIBILITY_PACKAGE' AND
                        A.packageId = B.packageId)

        )/*AND*/ AND

        /* Die Methode der Unterklasse ruft die Methode der Oberklasse auf, ergänzt sie
sozusagen */
        accesses.targetid = FB.id AND
        accesses.functionid = FA.id
;


CREATE Table VMethodsPerClass AS
```

```
/* Wie viele Methoden haben die Klassen, die vererbt werden, Konstruktoren und Desktruktoren
werden nicht mitgezählt */

SELECT
               type.id as TypeId,
               type.fullName as TypeName,
               count(distinct func.id) as numberOfMethods
FROM
               TTypes as type join
               TModelElements as elem on (type.id = elem.parentId) join
               TFunctions as func on (elem.parentid = func.classId) join
               TConstants as con on (elem.kindofelement = con.value) join
               TMembers as member on (member.id = func.id) join
               TConstants as constants on (member.visibility = constants.value)
WHERE
               elem.id = func.id and
               (con.name like 'FUNC_METHOD') AND
               member.isstatic = 0 AND
               member.isfinal = 0  AND
               constants.name != 'VISIBILITY_PRIVATE'
GROUP BY
               type.id,
               type.fullName

;


CREATE Table VMethodsInherited AS


SELECT
       type.id as TypeId,
       type.fullName as TypeName,
       SUM (mpc.numberOfMethods) as numberOfInheritedMethods
FROM
       TTypes as type join
       TInheritances as inheritance on (type.id = inheritance.classid) join
       VMethodsPerClass as mpc on (mpc.TypeId = inheritance.superid)
GROUP BY
       type.id,
       type.fullName

;

CREATE Table VOverridesPerClass AS

/* Wie viele Methoden der Klasse überschreiben eine Methode der Oberklasse */

SELECT
               type.id as TypeId,
               type.fullName as TypeName,
               count(distinct methodOverrides.OverrideMethodId) as numberOfOverrides
FROM
               TTypes as type join
               VMethodOverrides as methodOverrides on (type.id = methodOverrides.SubClassId)
GROUP BY
               type.id,
               type.fullName


;


/*

Eine Methode reimplementiert mehr als 50 % der Methoden einer Oberklasse,
ohne diese aufzurufen

*/

SELECT
       mi.TypeName as ClassName,
       mi.numberOfInheritedMethods * 1.0 as NumberOfInheritedMethods,
       opc.numberOfOverrides * 1.0 as NumberOfOverrides,
       (opc.numberOfOverrides*1.0) / (mi.numberOfInheritedMethods*1.0) as Ratio,
       files.pathname

FROM
       VMethodsInherited as mi join
```

```
        VOverridesPerClass as opc on (mi.TypeId = opc.TypeId) join
        TSourceEntities se on (mi.TypeId = se.id) join
        TFiles files on (se.sourcefileid = files.id)

WHERE
        (opc.numberOfOverrides*1.0) / (mi.numberOfInheritedMethods*1.0) > 0.5


;



DROP Table VMethodsInherited;
DROP Table VOverridesPerClass;
DROP Table VMethodsPerClass;
DROP Table VMethodOverrides;
```

Lösungsstrategie

Das Problem weist darauf hin, dass die Implementierung in der Unterklasse „zu niedrig“ in der Vererbungshierarchie platziert wurde. In diesem Fall ist die Unterklasse keine Spezialisierung der Oberklasse sondern eher eine Alternative bzw. Variante zu dieser. Die empfohlene Lösung besteht in folgenden Schritten:

1. Es wird eine gemeinsame Schnittstelle für Oberklasse und Unterklasse definiert
2. Die Unterklasse wird im Vererbungsbaum so verschoben, dass sie "parallel" zur Oberklasse liegt und beide von der neuen Schnittstelle erben.
3. Typereferenzen auf die Oberklasse werden durch Referenzen auf die neue Schnittstelle ersetzt

2.1.9 Gottdatei

Definition

Als Gottdateien werden Dateien bezeichnet, die insgesamt mehr als 2000 Brutto Lines of Codes enthalten.

Beschreibung

Kandidaten für Gottdateien sind zum einen Dateien, die Artefakte enthalten, welche bereits durch andere „Gott-Indikatoren“, z.B. Gottmethode, Gottklasse, aufgefallen sind. Daneben werden durch Gottdateien aber auch Dateien erfasst, die aus anderen Gründen sehr gross sind. Zum Beispiel Dateien mit mehreren (nicht öffentlichen) Klassen oder Dateien mit mehreren Methoden, deren Länge gerade unterhalb des Schwellwertes für Gottmethoden liegen. Derartige Dateien besitzen häufig zu viele Artefakte (die allerdings nicht durch andere Gott-Regeln erkannt werden), die daher separat nur mit überhöhtem Aufwand geändert, betrachtet und modifiziert werden können.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermittlung aller Dateien mit mehr als 2000 Brutto Lines of Code.
- Jede Datei, die dieses Kriterium erfüllt, ist eine Probleminstanz.

Erkennung mit SISSy

```
/* $Id: GottDatei.sql,v 1.2 2006/01/24 09:58:06 seng Exp $ */
/* FZI Forschungszentrum Informatik */


SELECT
        file.pathname,
```

```
        file.linesofcode
FROM
        TFiles  file
WHERE
        file.linesofcode > 2000;
```

Lösungsstrategie

Da dieses Problem meistens ein Zeichen für andere vorhandenen „Gott-Indikatoren“, z.B. Gottmethode, Gottklasse ist, sollten zunächst die dort angegebenen Lösungsvorschläge realisiert werden. Beseitigen diese das Problem Gottdatei nicht, sind folgende Lösungen denkbar:

- Dateien mit mehreren (nicht öffentlichen) Klassen aufspalten
- Ausfaktorisierung von inneren Klassen

2.1.10 Gottklasse (Attribut)

Definition

Eine Klasse deklariert selbst mehr als 50 Attribute (keine Konstanten), unabhängig von der Sichtbarkeit und des Typen der Attribue.

Beschreibung

Eine Klasse, die aus mehr als 50 Attributen besteht, kann nur schwer den konsistenten Zustand der Attribute sicherstellen. Auch das Verstehen und Warten derartiger Gottklassen ist mit einem deutlich überhöhten Aufwand verbunden. Die Erkennung kohäsiver Attribut-Teilmengen ist durch die fehlende Clusterung in mehrere Teilklassen erschwert.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermittelt werden alle Klassen, die mehr als 50 nicht-konstante Attribute deklarieren.
- Jede derartige Klasse stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* FZI */

CREATE OR REPLACE View VPublicAttributesPerClass AS

SELECT
        types.id as GottKlasseAttributID,
        types.fullname as GottKlasseAttribut,
        files.pathname as Dateiname,
        COUNT (variables.id) as AnzahlAttribute
FROM
        TTypes types join
        TVariables variables on (variables.classid = types.id) join
        TConstants cons on (variables.kindofvariable = cons.value) join
        TMembers members on (variables.id = members.id) join
        TConstants cons2 on (cons2.value = members.visibility) join
        TSourceEntities sourceEntities on (types.id = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id)
WHERE
        cons.name = 'VAR_FIELD'  AND
        variables.name <> '<self>' AND
```

```
        NOT (members.isstatic = 1 AND
                members.isfinal = 1)
GROUP BY
        types.id,
        types.fullname,
        files.pathname

;

SELECT
        *
FROM
        VPublicAttributesPerClass papc
WHERE
        papc.AnzahlAttribute > 50

;


DROP View VPublicAttributesPerClass;
```

Lösungsstrategie

Bevor man die Klasse restrukturiert, muss man die Ursache für die Entstehung des Problems identifizieren.
Die wichtigste und häufigste Ursache ist dass die Klasse semantisch nicht kohäsiv ist. In solchen Fällen wird man typischerweise versuchen kohäsive Gruppen (Cluster) von Attributen zu identifizieren und diese in neue Klassen oder Unterklassen der Klasse zu extrahieren. Hierzu nützliche Heuristiken sind die folgenden:

- Attribute die ähnliche Namen/Präfixe haben gehören wahrscheinlich zusammen. Zum Beispiel Attributen wie „depositAmount“ und „depositCurrency“ sind höchstwahrscheinlich verwandt [Fowler00].
- Attribute, die gemeinsam in Methoden referenziert werden gehören zusammen. Oft kann man aus mehreren Attributen mit einfachen Typen komplexere Datenstrukturen und neue Typen bilden. Diese sind dann gute Kandidaten für neue Klassen.
- Oft ist eine Zeitliche Analyse der Verwendung von Attributen sinnvoll. Zum Beispiel, kann es passieren, dass eine Klasse zu verschiedenen Zeitpunkten verschiedene Attribute verwendet. Dies deutet an einer mögliche Spezialisierung hin, und kann gute Hinweise für eine Extraktion von Unterklassen der Klasse liefern.

Man kann sich für eine dieser Heuristiken entscheiden und zunächst manuell die Klasse entsprechend aufteilen. Theoretisch besteht auch die Möglichkeit eine automatische Ballungsanalyse zu verwenden. Dafür ist es in der Regel notwendig, dass für jedes Attributpaar ein so genannten Ähnlichkeitsmaß definiert wird, das eine oder mehrere von den oben erwähnten Heuristiken berücksichtigen kann. Bei der Berücksichtigung von mehreren Heuristiken ist es empfehlenswert, eine Gewichtung der entsprechenden Komponenten im Ähnlichkeitsmaß zu vorzunehmen. In jeder Situation ist es weiterhin empfehlenswert, die automatisch vorgeschlagenen Clusterkandidaten manuell zu bestätigen.
Nachdem eine Aufteilung der Attribute vorliegt, muss man über die Kapselung dieser Attribute bzw. die Zuordnung von Methoden zu den zukünftigen Klassen nachdenken. Dazu kann man erneut auf automatischen Verfahren, wie Ballungsanalyse zurückgreifen. Wichtige Kriterien bei der Zuordnung von Methoden zu Klassen sind außer Zugriffe auf Attribute auch die Kopplung zwischen Methoden. Ziel soll sein, eine möglichst niedrige Kopplung zwischen den neu entstandenen Klassen zu haben.
Die zuverlässige Behebung einer Instanz vom Problem „Gottklasse“ ist in der Regel nicht voll-automatisch machbar. Aufgaben wie zum Beispiel Aufteilung von Methoden und Konstruktoren bei der Verschiebung von Attributen/Methoden verlangt große Domänenkenntnisse und technisches Wissen.

2.1.11 Gottklasse (Methode)

Definition

Eine Klasse/Interface deklariert (oder/und implementiert) mehr als 50 Methoden mit mindestens der Sichtbarkeit wie die umgebende Klasse/Interface.

Beschreibung

Eine Gottklasse besitzt aufgrund der vielen Methoden mit einer im Vergleich zur Klasse selbst großen Sichtbarkeit eine sehr breite Schnittstelle. Diese Breite führt dazu, dass sowohl deren Verwendung als auch Wartung überhöhte Aufwände bedeutet. Häufig besitzen derartige Klassen eine sehr geringe Kohäsion, d.h. die Klasse stellt mehr einen übergroßen Container von unterschiedlichen Abstraktionen und Funktionalität dar, als ein einheitliches Konzept, mit Daten und darauf zugreifenden Methoden.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Es werden alle Klassen und Interfaces ermittelt, die mehr als 50 Methoden deklarieren und/oder implementieren. Berücksichtigt werden dabei nur diejenigen Methoden, die mindestens so sichtbar wie die Klasse/das Interface sind. Dabei gilt public>protected>package-private>private.
- Jedes so ermittelte Artefakt stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
CREATE OR REPLACE View VPublicClassPublicMethods AS

SELECT
	types.id as GottKlasseAttributID,
	types.fullname as GottKlasseAttribut,
	files.pathname as Dateiname,
	COUNT (functions.id) as AnzahlMethoden
FROM
	TTypes types join
	TMembers typeMember on (types.id = typeMember.id) join
	TConstants cons3 on (typeMember.visibility = cons3.value) join
	TFunctions functions on (functions.classid = types.id) join
	TConstants cons on (functions.kindoffunction = cons.value) join
	TMembers members on (functions.id = members.id) join
	TConstants cons2 on (cons2.value = members.visibility) join
	TSourceEntities sourceEntities on (types.id = sourceEntities.id) join
	TFiles files on (sourceEntities.sourcefileid = files.id) join
	TSourceEntities sourceEntities2 on (sourceEntities2.id = functions.id) join
	TFiles files2 on (sourceEntities2.sourcefileid = files2.id)
WHERE
	(cons.name = 'FUNC_METHOD' OR
	cons.name = 'FUNC_CONSTRUCTOR' OR
	cons.name = 'FUNC_DESTRUCTOR') AND
	cons2.name = 'VISIBILITY_PUBLIC' AND
	cons3.name = 'VISIBILITY_PUBLIC'
GROUP BY
	types.id,
	types.fullname,
	files.pathname
;

CREATE OR REPLACE View VPackageClassPublicPackageMethods AS
SELECT
	types.id as GottKlasseAttributID,
	types.fullname as GottKlasseAttribut,
	files.pathname as Dateiname,
	COUNT (functions.id) as AnzahlMethoden
FROM
	TTypes types join
	TMembers typeMember on (types.id = typeMember.id) join
	TConstants cons3 on (typeMember.visibility = cons3.value) join
	TFunctions functions on (functions.classid = types.id) join
	TConstants cons on (functions.kindoffunction = cons.value) join
	TMembers members on (functions.id = members.id) join
	TConstants cons2 on (cons2.value = members.visibility) join
	TSourceEntities sourceEntities on (types.id = sourceEntities.id) join
	TFiles files on (sourceEntities.sourcefileid = files.id) join
```

```
        TSourceEntities sourceEntities2 on (sourceEntities2.id = functions.id) join
        TFiles files2 on (sourceEntities2.sourcefileid = files2.id)

WHERE
        (cons.name = 'FUNC_METHOD' OR
        cons.name = 'FUNC_CONSTRUCTOR' OR
        cons.name = 'FUNC_DESTRUCTOR') AND
        (cons2.name = 'VISIBILITY_PUBLIC' OR cons2.name = 'VISIBILITY_PACKAGE') AND
        cons3.name = 'VISIBILITY_PACKAGE'
GROUP BY
        types.id,
        types.fullname,
        files.pathname
;

SELECT
        *
FROM
        VPublicClassPublicMethods vpcpm
WHERE
        vpcpm.AnzahlMethoden > 50

UNION


SELECT
        *
FROM
        VPackageClassPublicPackageMethods vpcpm
WHERE
        vpcpm.AnzahlMethoden > 50

;


DROP View VPublicClassPublicMethods;

DROP View VPackageClassPublicPackageMethods;
```

Lösungsstrategie

Bevor man die Klasse restrukturiert, muss man die Ursache für die Entstehung des Problems identifizieren.
Die wichtigste und häufigste Ursache ist dass die Klasse semantisch unkohäsiv ist. In solchen Fällen wird man typischerweise versuchen kohäsive Gruppen (Cluster) von Methoden zu identifizieren und diese in neue Klassen oder Unterklassen der Klasse, zusammen mit den entsprechenden Attributen zu extrahieren. Hierzu nützliche Heuristiken sind folgenden:

- Methoden, die ähnliche Namen haben gehören wahrscheinlich zusammen.
- Methoden, die gemeinsame Attributen referenzieren gehören zusammen.
- Methoden, die durch Methodenaufrufe stark gekoppelt sind gehören mit einer höheren Wahrscheinlichkeit zusammen.
- Oft ist eine zeitliche Analyse der Verwendung von Methoden/Attributen sinnvoll. Zum Beispiel, kann es passieren, dass eine Klasse zu verschiedenen Zeitpunkten verschiedene Methoden/Attribute verwendet. Dies kann gute Hinweise für eine Extraktion von Unterklassen der Klasse liefern.

Man kann sich für eine dieser Heuristiken entscheiden und zunächst manuell die Klasse entsprechend aufteilen. Theoretisch besteht auch die Möglichkeit eine automatische Ballungsanalyse zu verwenden. Dafür ist es in der Regel notwendig, dass für jedes Methodenpaar ein so genanntes Ähnlichkeitsmaß definiert wird, das eine oder mehrere der oben erwähnten Heuristiken berücksichtigen kann. Bei der Berücksichtigung von mehreren Heuristiken ist es empfehlenswert eine Gewichtung der entsprechenden Komponenten im Ähnlichkeitsmaß vorzunehmen. In jeder Situation ist es weiterhin empfehlenswert, die automatisch vorgeschlagenen Clusterkandidaten manuell zu bestätigen.
Nachdem eine Aufteilung der Methoden vorliegt, muss man die Attribute zwischen den neuen Klassen verteilen. Dazu kann man erneut auf heuristikbasierte automatische Verfahren zurückgreifen.
Die zuverlässige Behebung einer Instanz vom Problem „Gottklasse“ ist in der Regel nicht voll-automatisch machbar. Aufgaben wie zum Beispiel die Aufteilung von Methoden und Konstruktoren bei der Verschiebung von Attributen/Methoden verlangt große Domänekenntnisse und technisches Wissen.

2.1.12 Gottmethode

Definition

Als Gottmethoden werden solche Methoden bezeichnet, deren Implementierung mehr als 200 Brutto lines of Code lang ist (gezählt einschliesslich der Zeile mit der öffnenden bis einschliesslich der Zeile der schließenden Klammer).

Beschreibung

Die Lesbarkeit und Verständlichkeit haben wesentlichen Einfluss auf die Wartbarkeit von Code. Nur Code, der verstanden wird, kann effizient modifiziert werden. Erstreckt sich eine Methode über mehrere Seiten (auf dem Bildschirm oder auf Papier), so ist die Lesbarkeit und Verstehbarkeit alleine aufgrund des Umfangs stark eingeschränkt.
Eine Gottmethode unterscheidet sich von einer „Labyrinthmethode“ (s.u.) dahingehend, dass letztere einen komplexen Kontrollfluss hat, jedoch nicht unbedingt sehr lang sein muss.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

- Ermittelt werden alle Methoden, deren Implementierung (von der öffnenden bis schließenden Klammer) mehr als 200 Brutto Lines of Code lang ist. Die „Klammerzeilen“ werden mit gezählt.
- Jede derartige Methode stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: GottMethode.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

SELECT
       P.id as PackageId,
       P.fullName as PackageName,
       C.id as ContainingClassId,
       C.name as ContainingClassName,
       func.id as FunctionId,
       func.name as FunctionName,
       func.numberOfLines as NumberOfLines,
       files.pathname as filename
FROM
       TFunctions func
INNER JOIN
       TModelElements elem on func.id = elem.id
LEFT OUTER JOIN
       TTypes C on elem.ParentId = C.id
LEFT OUTER JOIN
       TPackages P on
              C.PackageId = P.id
              OR elem.parentId = P.id
INNER JOIN
       TSourceEntities sourceEntities on (C.id = sourceEntities.id) join
       TFiles files on (sourceEntities.sourcefileid = files.id)
WHERE
       func.numberOfLines > 200 /* SCHWELLWERT */
ORDER BY
       func.numberOfLines DESC
;
```

Lösungsstrategie

Bei der Behebung von Gottmethoden ist das manuelle Eingreifen meistens unvermeidlich. In Abhängigkeit von der dahinterliegenden Ursache sind folgende alternative Strategien möglich:

- Wenn die Methode eine oder mehrere große „bedingte“ Anweisungen (z.B. „switch“ oder geschachtelte „if-else“) hat, und wenn andere Methoden in der selben Klasse ähnliche Anweisungen enthalten, ist das ein Hinweis darauf, dass Polymorphie angebracht ist. Das heißt, dass jeder Zweig in den komplexen bedingten Anweisungen eine Spezialisierung der Klasse darstellen könnte. In diesem Fall, besteht die Lösung darin, dass diese alternativen Zweige in spezialisierenden Methoden von entsprechenden Unterklassen extrahiert werden.
- In der Mehrheit der Situationen besteht die Lösung darin, dass man aus der Methode neue Methoden (Helfermethoden) extrahiert („extract method“ Restrukturierung aus [Fowler00]). Eine nützliche und sehr effektive Heuristik, die man hier verwenden kann, ist, dass Methodenfragmente, die gesondert kommentiert sind, gute Kandidaten für Helfermethoden sind. Die Namen der neu-extrahierten Methoden können auf Basis dieser Kommentare festgelegt werden. Wenn die Methode besonders viele temporäre Variablen bzw. Parameter hat, und wenn das auch semantisch begründet ist, kann man ein so genanntes Methodenobjekt (eigentlich eine neue Klasse) extrahieren. Die ursprüngliche Methode und die extrahierten Helfermethoden werden dabei in eine neue Klasse verschoben, und die alten Variablen und Methodenparameter werden Attribute der neuen Klasse.

2.1.13 Gottpaket

Definition

In einem Paket bzw. einem Verzeichnis sind mehr als 50 öffentliche, nichtgeschachtelte Klassen oder Interfaces deklariert.

Beschreibung

Pakete/Verzeichnisse sind ein Mittel zur Bildung einer über der Ebene von Klassen und Interfaces liegenden weiteren Abstraktionsschicht innerhalb von Softwaresystemen. Enthält ein Paket/Verzeichnis sehr viele Klassen/Interfaces, besteht die Gefahr, dass diese Abstraktion nicht mehr mit vertretbarem Aufwand verstanden und gewartet werden kann, da der Inhalt zu umfangreich und daher vermutlich eine weitere Dekomposition angebracht scheint.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Architekturebene

Informale Erkennung

- Ermittelt werden alle Pakete/Verzeichnisse, die mehr als 50 öffentliche, nicht-geschachtelte Klassen/Interfaces deklarieren.
- Jedes derartige Paket/Verzeichnis stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: GottPaket.sql,v 1.1 2006/01/18 15:25:32 seng Exp $ */
/* FZI Forschungszentrum Informatik */

create or replace view VPackElements as
SELECT
       pack.id as PackageId,
       pack.fullName as PackageName,
       elem.id as ElemId
FROM
       TPackages as pack join
       TTypes as elem on (pack.id = elem.packageId) join
```

```
        TMembers as member on (elem.id = member.id) join
        TConstants as con on (member.visibility = con.value) join
        TSourceEntities as se on (elem.id = se.id)
WHERE
        se.sourcefileid <> '-1' AND
        elem.classid = '-1' /* no inner classes*/ AND
        con.name = 'VISIBILITY_PUBLIC'
;


/* Count directly contained model elements */
SELECT
        res.PackageId as PackageId,
        res.PackageName as PackageName,
        count(distinct res.ElemId) as noContainedElements
FROM
        VPackElements as res
GROUP BY
        res.PackageId,
        res.PackageName
HAVING
        count(distinct res.ElemId) > 50
ORDER BY
        noContainedElements
;


drop view VPackElements;
```

Lösungsstrategie

Die Lösungsstrategie für dieses Problem besteht in der Aufteilung des Gottpakets, meistens in Unterpakete nach funktionalen Gesichtspunkten. Eine neue Einteilung kann manuell bestimmt werden, oder aber automatisch z.B. mit Hilfe von Clustering-Algorithmen oder genetischen Algorithmen bestimmt werden. Dabei wird ein Kopplungsmaß zwischen Klassen des Pakets definiert und berechnet, das die wichtigste Rolle in einer Similaritätsmaß bzw. Kostenfunktion spielt. Um eine gewisse semantische Konsistenz zu bewahren ist es empfehlenswert die automatisch vorgeschlagenen Cluster vom Entwickler manuell bestätigen zu lassen.

2.1.14 Halbherzige Operationen

Definition

Zusammenhängende Methodencluster werden unvollständig bzw. inkonsistent implementiert, wodurch die dem Methodencluster zugrundeliegende Funktionalität falsch werden kann. In JAVA sind dies `Object.equals()` und `Object.hashCode()`. In C++ sind dies `operator++(void)` und `operator++(int)` (inkl. die Minus-Varianten), der Copy-Konstruktor und der operator= sowie die Forderung nach virtuellen Destruktoren bei der Existenz einer virtuellen Methode.

Beschreibung

Bestimmte Methoden von Klassen müssen bezüglich ihrer Implementierung konsistent zueinander gehalten werden, da Klienten der Klassen auf diese Konsistenz vertrauen. Wird das Verhalten einzelner Methoden durch neue Implementierungen oder die Verwendung unterschiedlicher Bindungen (virtual vs. nicht virtual) verändert, kann diese Konsistenz zerstört werden, d.h. die der Methodengruppe zugrunde liegende Funktionalität kann hierdurch falsch werden.
Während die Konsistenz der Implementierung von Anwendungscode nicht allgemein auszudrücken ist, gibt es jedoch seitens der Programmiersprachen folgende Grundregeln, die beachtet werden sollten:

Java

- Object.equals() und Object.hashCode() konsistent halten, weil Collection-Klassen mittels hashCode zunächst "Äquivalenzkandidaten" identifizieren.

C++

- operator++(void) und operator++(int) (bzw. operator--) zusammen überschreiben, um eine Konsistenz zwischen Post- und Präfixoperatoren sicherzustellen.

- Copy-Konstruktor und operator=
- Wenn eine Methode "virtual" ist, muss auch der Destruktor "virtual" sein.

Dies bedeutet jeweils, dass die Änderung einer dieser Methoden die Änderung der anderen implizieren muss.

Delphi

- Konstruktor und Destruktor sollten beide überschrieben werden

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

Hinweis: Nachfolgende Suchmuster werden sprachspezifisch ausgeführt:

JAVA:

- Suche aller Klassen, die entweder equals() oder hashCode() implementieren, nicht jedoch beide.

C++: Suche aller Klassen, die

- jeweils nur einen der beiden Operatoren operator++(void) und operator++(int) überschreiben,
- jeweils nur einen der beiden Operatoren operator--(void) und operator --(int) überschreiben,
- entweder den Copy-Konstruktor oder den operator= implementieren, nicht jedoch beide.
- einen nicht-virtuellen Destruktor haben, obwohl virtuelle Methoden deklariert werden.

Jede Verletzung wird separat gezählt, auch mehrere innerhalb ein und derselben Klasse.

Delphi:

- suche nach Klassen, die den Konstruktor aber nicht den Destruktor überschreiben (Delphi)

Erkennung mit SISSy

```
/* $Id: HalbherzigeOperationen.sql,v 1.1 2006/01/18 15:25:32 seng Exp $ */
/* FZI Forschungszentrum Informatik */
/*
*/
SELECT
        types.name,
        files.pathname
FROM
        TTypes types join
        TFunctions functions on (types.id = functions.classid) join
        TSignatures sig on (sig.functionid = functions.id) join
        TSourceEntities sourceEntities on (types.id = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id) join
        TAccesses returnaccess on (functions.returntypedeclarationid = returnaccess.id) join
        TTypes returnType on (returnaccess.targetid = returnType.id)
WHERE
        (sig.signature = 'equals(java.lang.Object)' AND
        returnType.name = 'boolean')
        OR
        (sig.signature = 'hashCode()' AND
        returnType.name = 'int')

EXCEPT

SELECT
        types.name,
        files.pathname
FROM
        TTypes types join
        TFunctions meth1 on (types.id = meth1.classid) join
        TSignatures sig1 on (sig1.functionid = meth1.id) join
        TAccesses returnaccess1 on (meth1.returntypedeclarationid = returnaccess1.id) join
        TTypes returnType1 on (returnaccess1.targetid = returnType1.id) join
        TFunctions meth2 on (types.id = meth2.classid) join
```

```
        TSignatures sig2 on (sig2.functionid = meth2.id) join
        TAccesses returnaccess2 on (meth2.returntypedeclarationid = returnaccess2.id) join
        TTypes returnType2 on (returnaccess2.targetid = returnType2.id) join
        TSourceEntities sourceEntities on (types.id = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id)

WHERE
        sig1.signature = 'equals(java.lang.Object)'
        AND returnType1.name = 'boolean'
        AND sig2.signature = 'hashCode()'
        AND returnType2.name = 'int'
;
```

Lösungsstrategie

Da dieses Problem vielmehr mit der Semantik der Anwendung als mit ihrer Struktur zusammenhängt kann man hier nur eine allgemeine Aussage machen. Mögliche Kandidaten von inkonsistenten Operationen müssen manuell überprüft und durch Nachimplementieren beseitigt werden.

2.1.15 Identitätsspaltung

Definition

Mehrere öffentliche Klassen und/oder Schnittstellen (Interface) innerhalb eines Systems haben den gleichen Namen. Etwaige Unterschiede in der Groß- und Kleinschreibung werden hierbei als Unterscheidungsmerkmal ignoriert.

Beschreibung

Klassen und Schnittstellen werden im Code in den meisten Fällen nicht über ihren vollqualifizierten Namen (z.B. `java.util.HashMap`) referenziert, sondern nur über ihren Namen (z.B. `HashMap`). Gibt es nun im System mehrere Klassen/Schnittstellen mit dem gleichen Namen, besteht eine hohe Verwechslungsgefahr, da die tatsächlich verwendete Klasse/Schnittstelle nicht sofort ersichtlich ist. Stattdessen muss die verwendete Klasse durch die Analyse des Kontextes (der importierten Klassen/Pakete) bestimmt werden. Auch die Verwendung moderner IDEs beseitigt das Problem nicht vollständig. Diese können zwar den vollqualifizierten Namen automatisch ermitteln, zeigen ihn aber oft nur in Form von Tooltips an. Beim einfachen Lesen des Codes (oder in Ausdrucken) fehlt diese Information aber.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermittelt werden Gruppen von öffentlichen Klassen und Interfaces, die alle (unabhängig von der Groß- und/oder Kleinschreibung) den gleichen Namen besitzen.
- Jede derartige Gruppe stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/*
 * $Id: Identitaetsspaltung.sql,v 1.1 2006/01/18 15:25:31 seng Exp $
 */

/*

Namensvergleich ignoriert Groß- und Kleinschreibung

*/
SELECT
```

```
        lower(type.name) AS Class
        , COUNT(type.id) AS DuplicateCount
FROM
        TTypes type join
        TMembers member on (type.id = member.id) join
        TConstants con on (member.visibility = con.value) join
        TSourceEntities sourceEntities on (type.id = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id)
WHERE
        /* Anonyme Klassen sind nicht von Interesse */
        type.name <> '' AND    /* Nur öffentliche Typen */
        con.name = 'VISIBILITY_PUBLIC' AND
        /* Keine inneren Klassen */
        type.classid = '-1'
GROUP BY
        lower(type.name)
HAVING COUNT(type.id) > 1
;
```

Lösungsstrategie

Gleichnamige Klassen bzw. Schnittstellen müssen umbenannt werden. Wenn dieses Problem nicht nur vereinzelt vorkommt ist es empfehlenswert eine Ursachenforschung zu machen. Bei ganzen duplizierten Verebungshierarchien zum Beispiel kommen eventuell tiefgreifendere Restrukturierungsmaßnahmen in Betracht.

2.1.16 Importchaos

Definition

Ein Importchaos liegt vor, wenn die aufgeführten Imports bzw. Includes redundant oder nicht explizit vorliegen. Für die einzelnen Sprachen bedeutet dies:

- JAVA
  1. Importieren einer Klasse aus dem eigenen Paket
  2. Importieren einer Klasse aus `java.lang`
  3. mehrfaches Importieren ein und derselben Klasse.
  4. On-demand-Imports von gesamten Paketen
- C++.
  1. Mehrfaches direktes Includieren ein und derselben Datei

Beschreibung

Die Qualität der Imports/Includes ist wichtig, um den benutzten Kontext eines Programmfragmentes zügig aufspannen zu können. Liegt diese Qualität nicht vor, so verhindert das zwar nicht einen erfolgreichen Compile-Schritt, verhindert aber das effektive Nutzen der Import/Include-Statements für einen ersten semantischen Überblick. Je häufiger die Imports/Includes redundant, überflüssig oder allgemein sind, desto weniger helfen sie.

Programmiersprache

C++, Java, C#, ~~Delphi~~

Betrachtungsebene

Designebene

Informale Erkennung

Die Probleminstanzen ergeben sich entsprechend folgender Analysen:

- Ermittelt werden Imports/Includes, die durch mehrere Import/Include-Anweisungen in eine Datei eingefügt werden. Jeder mehrfache Import/Include ist eine Probleminstanz.
- Ermittlung von java.lang Importen. Jede entsprechende Import Anweisung ist eine Probleminstanz.
- Ermittlung von On-Demand-Imports durch Suche nach ‚*' am Ende des Imports. Jedes entsprechende Import-Anweisung stellt eine Probleminstanz dar.
- Ermittlung von Importen von Klassen, die im Verzeichnis der analysierten Datei existieren (überflüssige Importanweisung). Jede entsprechende Importanweisung stellt ein Probleminstanz dar.
- Jede Import/Include Anweisung kann maximal eine Probleminstanz darstellen.

Erkennung mit SISSy

```
/* $Id: ImportChaos.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */




/*

Definition:

Ein Importchaos liegt vor, wenn die aufgeführten Imports bzw. Includes
nicht redundant oder nicht explizit vorliegen. Für die einzelnen
Sprachen bedeutet dies:

Für JAVA:
 - Importieren einer Klasse aus dem eigenen Paket (nicht möglich zu prüfen)
 - Importierung einer Klasse aus java.lang (macht keinen Sinn)
 - mehrfaches Importieren ein und derselben Klasse.
 - On-demand-Imports von gesamten Paketen.

Für C++:
 - Mehrfaches direktes Includieren ein und derselben Datei.

Beschreibung:

Es sollte kein Importchaos geben, da Imports wichtig sind, um den
benutzten Kontext eines Programmfragmentes zügig aufspannen zu können.
Damit müssen Imports auch eine gewisse Qualität aufweisen, die bei Redundanz
aufgrund der unnötig großen Import-Blöcke bzw. bei on-demand-Imports aufgrund
der fehlenden Transparenz nicht gegeben ist.

Implementierung:



*/
select distinct *
from
        (
        select
                f.id as SourceFileId,
                f.pathName as SourceFileName,
                con.name as KindOfTarget,
                target.id as TargetId,
                target.pathName as TargetName,
                count (*) as ImportCount
        from
                TFiles f,
                TImports imp,
                TConstants con,
                TFiles target
        where
                imp.fileId = f.id and
                imp.kindOfTarget = con.value and
                target.id = imp.targetId
        group by
                SourceFileId,
                SourceFileName,
                con.name,
                target.id,
                TargetName

        union

        select
                f.id as SourceFileId,
```

```
                f.pathName as SourceFileName,
                con.name as KindOfTarget,
                target.id as TargetId,
                target.fullName as TargetName,
                count (*) as ImportCount
        from
                TFiles f,
                TImports imp,
                TConstants con,
                TPackages target
        where
                imp.fileId = f.id and
                imp.kindOfTarget = con.value and
                target.id = imp.targetId
        group by
                SourceFileId,
                SourceFileName,
                con.name,
                target.id,
                TargetName

        union

        select
                f.id as SourceFileId,
                f.pathName as SourceFileName,
                con.name as KindOfTarget,
                target.id as TargetId,
                target.fullName as TargetName,
                count (*) as ImportCount
        from
                TFiles f,
                TImports imp,
                TConstants con,
                TTypes target
        where
                imp.fileId = f.id and
                imp.kindOfTarget = con.value and
                target.id = imp.targetId
        group by
                SourceFileId,
                SourceFileName,
                con.name,
                target.id,
                TargetName

        union

        select
                f.id as SourceFileId,
                f.pathName as SourceFileName,
                con.name as KindOfTarget,
                target.id as TargetId,
                pack.fullName || '.' || target.name as TargetName,
                count (*) as ImportCount
        from
                TFiles f,
                TImports imp,
                TConstants con,
                TPackages pack,
                TVariables target
        where
                imp.fileId = f.id and
                imp.kindOfTarget = con.value and
                target.id = imp.targetId and
                target.packageId = pack.id
        group by
                SourceFileId,
                SourceFileName,
                con.name,
                target.id,
                TargetName

        union

        select
                f.id as SourceFileId,
                f.pathName as SourceFileName,
                con.name as KindOfTarget,
                target.id as TargetId,
```

```
                pack.fullName || '.' || sig.signature as TargetName,
                count (*) as ImportCount
        from
                TFiles f,
                TImports imp,
                TConstants con,
                TPackages pack,
                TFunctions target,
                TSignatures sig
        where
                imp.fileId = f.id and
                imp.kindOfTarget = con.value and
                target.id = imp.targetId and
                target.packageId = pack.id and
                target.id = sig.functionId and
                con.name != 'FUNC_TYPE_DELEGATE'
        group by
                SourceFileId,
                SourceFileName,
                con.name,
                target.id,
                TargetName
        ) as res
where
        ImportCount > 1 or
        KindOfTarget LIKE 'PACK_%' or
        targetname LIKE 'java.lang%'

UNION

/* Imports aus eigenem Paket */

select
        f.id as sourcefileid,
        f.pathname as sourcefilename,
        'TYPE_CLASS',
        type2.id as targetid,
        type2.name as targetname,
        '0'
from
        TFiles f join
        TSourceEntities s on (s.sourcefileid = f.id) join
        TTypes type on (s.id = type.id) join
        TTypes type2 on (type.packageid = type2.packageid and type.id <> type2.id) join
        TImports imp on (imp.fileid = f.id and imp.targetid = type2.id)

;
```

Lösungsstrategie

Entfernen der überflüssigen Include-, Using- oder Import-Statements.

2.1.17 Importlüge

Definition

Eine Importlüge liegt vor, wenn eine Klasse bzw. Datei oder ein komplettes Paket importiert/inkludiert wird, ohne dass die importierten Klassen/ Dateien benutzt werden, d.h. zum Compilieren sind die entsprechenden Klassen/ Dateien nicht notwendig.

Beschreibung

Einzelne Klassen oder ganze Pakete (bzw. Bibliotheken) werden in den Quellcode eingebunden, obwohl diese gar nicht referenziert werden. Diese unnötigen Importe verändern zwar das Verhalten des Systems nicht, erhöhen aber die Anforderungen an den Build-Prozess. Alle aufgeführten Klassen bzw. Pakete müssen für einen erfolgreichen Build verfügbar sein. Außerdem entsteht durch eine Importlüge ein falsches Bild des Systems, da ein falscher Kontext suggeriert wird.

Programmiersprache

C++, Java, C#, ~~Delphi~~

Betrachtungsebene

Codeebene

Informale Erkennung

Eine Verletzung liegt für die Zählung vor, wenn

- beim Import von konkreten Klassen bzw. Dateien, diese nicht verwendet werden
- beim Import von Paketen (hier auch Bibliotheken, d.h. nicht selbst geschriebene Systemteile) kein Artefakt daraus benutzt wird.

Die Verletzungsanzahl ergibt sich aus der Nummer der beiden Einzelzählungen

Erkennung mit SISSy

```
/* $Id: ImportLuege.sql,v 1.1 2006/01/18 15:25:32 seng Exp $ */
/* FZI Forschungszentrum Informatik */


create table VKnowsRel as

select distinct
       res.classId,
       res.className,
       res.knownClassId,
       res.knownClassName,
       res.isStatic
from
       (
       /* Type accesses including declaration type accesses */
       select distinct
              cls.id as ClassId,
              cls.fullName as ClassName,
              knowncls.id as KnownClassId,
              knowncls.fullName as KnownClassName,
              con.name = 'TYPEACCESS_STATIC' as IsStatic
       from
              TConstants con,
              TAccesses acc,
              TTypes cls,
              TTypes knowncls
       where
              con.name like 'TYPEACCESS_%' and
              con.value = acc.kindOfAccess and
              acc.classId = cls.id and
              acc.targetId = knowncls.id

       union

       /* Accesses to members */
       select distinct
              cls.id as ClassId,
              cls.fullName as ClassName,
              knowncls.id as KnownClassId,
              knowncls.fullName as KnownClassName,
              dm.isStatic = 1 as IsStatic
       from
              TTypes as cls,
              TAccesses acc,
              TMembers as dm,
              TTypes knowncls
       where
              cls.id = acc.classId and
              acc.targetId = dm.id and
              dm.classId = knowncls.id
       ) as res,
       TTypes as types,
       TConstants as c
where
       res.classId != res.knownClassId and
       res.knownClassId = types.id and
       types.kindOfType = c.value and
```

```
        (c.name like '%CLASS' or
                c.name like '%INTERFACE')
;


/*
The types, the class imports
*/

(SELECT
        file.pathname as File,
        importedType.id as unusedimportid,
        importedType.fullname as unuseditem
FROM
        TFiles file join
        Timports import on (import.fileid = file.id) join
        TTypes importedType on (import.targetid = importedType.id)

EXCEPT

/* The Types the class really needs */
SELECT
        file.pathname,
        knowsRel.knownclassid,
        knowsRel.knownclassname
FROM
        TFiles file join
        TSourceEntities sourceEntities on (file.id = sourceEntities.sourcefileid) join
        TTypes types on (types.id = sourceEntities.id) join
        VKnowsRel knowsRel on (knowsRel.classid = types.id)
)

UNION
(
/* All packages a file imports */
SELECT  DISTINCT
        file.pathname,
        file.id,
        importedPackages.fullname as unuseditem
FROM
        TFiles file join
        Timports import on (import.fileid = file.id) join
        TPackages importedPackages on (importedPackages.id = import.targetid)

EXCEPT

/* The packages, that are actually required */
SELECT  DISTINCT
        file.pathname,
        file.id,
        importedPackages.fullname as usedPackageImport
FROM
        TFiles file join
        TSourceEntities source on (file.id = source.sourcefileid) join
        TTypes classInFile on (classInFile.id = source.id) join
        Timports import on (import.fileid = file.id) join
        TPackages importedPackages on (importedPackages.id = import.targetid) join
        TTypes importedType on (importedPackages.id = importedType.packageid) join
        VKnowsRel knowsRel on (knowsRel.knownClassId = importedType.id)
WHERE
        knowsRel.classid = classInFile.id

)
;

DROP TABLE VKnowsRel;
```

Lösungsstrategie

Nicht benötigte Import-Anweisungen können einfach entfernt werden.

2.1.18 Informelle Dokumentation

Definition

Eine informelle Dokumentation liegt vor, wenn eine öffentliche Methodendeklaration keinen formalen Dokumentationsmechanismus für deren Dokumentation verwendet. Da alle verbreiteten Kommentierungsformalismen (insbesondere die verbreiteten Werkzeuge Javadoc und Doxygen) als Beginn die Zeichenkette `/**` verwenden, wird eine informelle Dokumentation daran festgemacht, dass vor einer öffentlichen Methodendeklaration kein durch `/**` eingeleiteter Kommentar existiert. Um keinen Kommentierungsmechanismus zu bevorzugen, werden keinerlei Tags überprüft.

Beschreibung

Mechanismen zur Quelltextdokumentation - wie z.B. Javadoc - bieten einen einheitlichen, bekannten Standard, mit dessen Hilfe der Code dokumentiert und zum Beispiel eine gut lesbare API-Dokumentation in Form von HTML-Seiten erzeugt werden kann. Neuere IDEs wie z.B. Eclipse können derartige Dokumentations-Skelette bereits automatisch erzeugen. Halten sich die Programmierer nicht an die vorgegebenen Mechanismen, bzw. werden keine solchen Mechanismen vorgegeben, kann es keine überschaubare, umfassende, mit Werkzeugen bearbeitbare und aktuelle Dokumentation geben.

Programmiersprache

C++, Java, C#, ~~Delphi~~

Betrachtungsebene

Codeebene

Informale Erkennung

- Es müssen alle öffentlichen Methodendeklarationen ermittelt werden, die keinen formalen Kommentar (/**) besitzen.
- Jede so ermittelte Methodendeklaration stellt eine Probleminstanz dar.
- Eine Prüfung, ob automatisch erzeugte Kommentarrümpfe auch mit sinnvollem Inhalt gefüllt wurden ist werkzeuggestützt nicht möglich.

Erkennung mit SISSy

```
/* $Id: InformelleDokumentation.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

CREATE OR REPLACE VIEW VMethodsWithoutFormalComment as

SELECT
        function.id,
        function.name,
        function.classid
FROM
        TFunctions function join
        TMembers member on (function.id = member.id) join
        TConstants con on (member.visibility = con.value)
WHERE
        con.name = 'VISIBILITY_PUBLIC'

EXCEPT

SELECT
        function.id,
        function.name,
        function.classid
FROM
        TFunctions function join
        TAnnotations annotation on (function.id = annotation.modelelementid) join
        TComments comments on (annotation.annotationid = comments.id) join
        TSourceEntities functionSE on (function.id = functionSE.id) join
        TSourceEntities commentSE on (commentSE.id = comments.id)
WHERE
```



```
        comments.isformal > 0 AND
        /* Heuristik: Formaler Kommentar muss vor der Methode stehen */
        /* Siehe Suns Seiten zu JavaDoc: A doc comment is written in HTML and must precede a
class, field, constructor or method declaration */
        commentSE.startline < functionSE.startline
;

SELECT
        vmwfc.id,
        vmwfc.name,
        type.fullname,
        file.pathname
FROM
        VMethodsWithoutFormalComment vmwfc join
        TTypes type on (vmwfc.classid = type.id) join
        TSourceEntities sourceentity on (vmwfc.id = sourceentity.id) join
        TFiles file on (sourceentity.sourcefileid = file.id)

;

DROP VIEW VMethodsWithoutFormalComment;
```

Lösungsstrategie

Es können mit Werkzeugen Kommentarrahmen erzeugt werden, um den Programmieren den Umgang mit den Dokumentationswerkzeugen zu erleichtern. Diese müssen anschließend alle öffentlichen Methoden und Klassen entsprechend dokumentieren.

2.1.19 Interface-Bypass

Definition

Eine durch ein Interface oder eine abstrakte Klasse bereitgestellte öffentlich sichtbare Methode wird von einem außerhalb der Vererbungshierarchie stehenden Klienten nicht über diese Abstraktion sondern direkt über eine (in-)direkte konkrete Unterklasse angesprochen, d.h. die Programmierung eines bestimmten Klienten verursacht einen Interface-Bypass.

Beschreibung

Klienten verwenden statt einer Abstraktion in Form eines Interfaces oder einer abstrakten Klasse direkt eine Implementierung der Abstraktion, obwohl die angeforderte Funktionalität über die Abstraktion verfügbar ist. Kann die Deklaration einer öffentlich sichtbaren Methode keiner Abstraktion eindeutig zugeordnet werden (z.B. bei aus mehreren Interfaces und abstrakten Klassen zusammengesetzten Abstraktionen), liegt eine Verletzung im Sinne des Problemmusters nur dann vor, wenn keine dieser zusammengesetzten Abstraktion verwendet wird. Durch Interface-Bypässe entsteht eine ungewollte Abhängigkeit, da die Klienten von den konkreten Klassen abhängen, was die Wiederverwendung deutlich erschwert. Zugriffe auf die Implementierung durch Unterklassen derselbigen stellen keine Verletzung dar, da hier sowieso schon eine Abhängigkeit durch die Vererbungsbeziehung besteht.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Gezählt werden Methodenpaare von aufrufender und aufgerufener Methode.
- Die aufgerufene Methode muss in einer Abstraktion A deklariert sein, der konkrete Aufruf erfolgt allerdings zu einer Implementierung dieser Methode.

- Nicht betrachtet werden diejenigen aufrufenden Methoden, deren umgebende Klasse mit A in einer Vererbungsbeziehung steht.

Erkennung mit SISSy

```
/* $Id: InterfaceBypass.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */


SELECT
        accessedFunction.id,
        accessedFunction.name as AccessedFunctionName,
        functionType.name as AccessedFunctionBelongsToType,
        baseType.name as preferredType,
        accessedFromFunction.id as AccessedFromFunctionId,
        accessedFromFunction.name as AccessedFromFunctionName,
        accessedFromFunctionFile.pathname
FROM
        TAccesses accesses join
        TMembers members on (accesses.targetid = members.id) left join
        TFunctions baseFunction on (members.overridenmemberid = baseFunction.id) join
        TMembers basemember on (members.overridenmemberid = basemember.id) join
        TFunctions accessedFunction on (members.id = accessedFunction.id) join
        TTypes functionType on (accessedFunction.classid = functionType.id) join
        TTypes baseType on (baseFunction.classid = baseType.id) join
        TFunctions  accessedFromFunction  on  (accesses.functionid  =  accessedFromFunction.id)
join
        TSourceEntities      accessedFromFunctionSE      on      (accessedFromFunction.id      =
accessedFromFunctionSE.id) join
        TFiles      accessedFromFunctionFile      on      (accessedFromFunctionFile.id      =
accessedFromFunctionSE.sourcefileid)
WHERE
        accesses.targetid = members.id AND
        members.overridenmemberid <> -1 AND
        NOT EXISTS (
                SELECT *
                FROM TInheritances i
                WHERE i.classId = accesses.classid AND basemember.classid = i.superId
        )
GROUP BY
        accessedFunction.id,
        accessedFunction.name,
        functionType.name,
        baseType.name,
        accessedFromFunction.id,
        accessedFromFunction.name,
        accessedFromFunctionFile.pathname
```

Lösungsstrategie

Häufige Ursachen für Interface Bypass sind:

- Inkonsistente Verwendung des Vererbungsbaums durch Evolution: es kann passieren dass die abstrakte Schnittstelle, die die Abstraktion definiert nach der Implementierung des Klienten definiert wurde (d.h. der Vererbungsbaum wurde „nach oben“ erweitert), und Klienten wurden nicht mehr aktualisiert. In solchen Fällen besteht die Lösung darin, dass alle Verwendungsstellen in Klienten angepasst werden.
- Vererbung statt Komposition: die Vererbungsbeziehung zwischen Abstraktion und Unterklasse ist nicht gerechtfertigt. Vererbung wird häufig fälschlicherweise statt Komposition verwendet um schnell Implementierung wieder zu verwenden. In C++ ist diese Situation in den Fällen wo „private“ Vererbung verwendet wird offensichtlich. Die Lösung besteht in diesem Fall in der Ersetzung der Vererbungsbeziehung durch Komposition.

2.1.20 Klasseninzest

Definition

Eine Oberklasse bzw. ein Interface darf keinerlei Wissen über ihre direkten oder indirekten Unterklassen bzw. Unter-Interfaces haben, d.h. sie darf weder Attribute und Methoden der Unterklassen ansprechen, noch darf sie den konkreten Typ der Unterklasse bzw. des Unter-Interfaces (z.B. für Rückgabewerte) kennen.

Beschreibung

Eine Oberklasse bzw. Interface darf keine Abhängigkeiten zu ihren Unterklassen bzw. Unter-Interfaces haben, da solche Abhängigkeiten in manchen Situationen dazu führen können, dass Änderungen in der Verallgemeinerung notwendig sind sobald eine neue Konkretisierung definiert wird. Ein typisches Beispiel für eine solche Abhängigkeit sind Typabfragen mit Hilfe von "instanceof" in der Verallgemeinerung, so dass die Konkretisierung zur Laufzeit in Abhängigkeit des tatsächlich vorliegenden Objekts ein anderes Verhalten aufweisen kann. Diese Abhängigkeiten können z.B. durch Polymorphie, durch Änderung der Vererbungsbeziehungen oder durch konsequente Verwendung von Schnittstellen aufgelöst werden.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Es werden alle Oberklasse - Unterklasse Paare ermittelt, für die gilt, dass die Oberklasse die Unterklasse „kennt“. Das heißt, dass zur Übersetzung der Oberklasse die Unterklasse benötigt wird.
- In obiger Beschreibung kann für Ober- bzw. Unterklasse jeweils auch Interface eingesetzt werden.
- Jedes so ermittelte Paar stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: KlassenInzest.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:

Eine Oberklasse darf keinerlei Wissen über ihre direkten oder indirekten
Unterklassen haben, d.h. sie darf weder Attribute und Methoden der Unterklassen
ansprechen, noch darf sie den konkreten Typ der Unterklasse (z.B. für
Rückgabewerte) kennen.

Beschreibung:

Eine Oberklasse soll keine Abhängigkeiten zu ihren Unterklassen haben, da solche
Abhängigkeiten in manchen Situationen dazu führen können, dass Änderungen in der
Oberklasse notwendig sind sobald eine neue Unterklasse definiert wird. Ein
typisches Beispiel für eine solche Abhängigkeit sind Typabfragen mit Hilfe von
"instanceof" in der Oberklasse, so dass die Oberklasse zur Laufzeit in
Abhängigkeit des tatsächlich vorliegenden Objekts ein anderes Verhalten aufweisen
kann. Diese Abhängigkeiten können z.B. durch Polymorphie, durch Änderung der
Vererbungsbeziehungen oder durch konsequente Verwendung von Schnittstellen
aufgelöst werden.

Implementierung:

Statische Typzugriffe und Zugriffe zu statischen Merkmale sind ignoriert. Solche
Fälle sind Instanzen von VerboteneKlassenLiebe, aber nicht KlassenInzest.

(OS) Auch statische Abhängigkeiten sollten berücksichtigt werden. Anfrage geändert


*/
```

```
create table VKnowsRel as

select distinct
       res.classId,
       res.className,
       res.knownClassId,
       res.knownClassName,
       res.isStatic
from
       (
       /* Type accesses including declaration type accesses */
       select distinct
              cls.id as ClassId,
              cls.fullName as ClassName,
              knowncls.id as KnownClassId,
              knowncls.fullName as KnownClassName,
              con.name = 'TYPEACCESS_STATIC' as IsStatic
       from
              TConstants con,
              TAccesses acc,
              TTypes cls,
              TTypes knowncls
       where
              con.name like 'TYPEACCESS_%' and
              con.value = acc.kindOfAccess and
              acc.classId = cls.id and
              acc.targetId = knowncls.id

       union

       /* Accesses to members */
       select distinct
              cls.id as ClassId,
              cls.fullName as ClassName,
              knowncls.id as KnownClassId,
              knowncls.fullName as KnownClassName,
              dm.isStatic = 1 as IsStatic
       from
              TTypes as cls,
              TAccesses acc,
              TMembers as dm,
              TTypes knowncls
       where
              cls.id = acc.classId and
              acc.targetId = dm.id and
              dm.classId = knowncls.id
       ) as res,
       TTypes as types,
       TConstants as c
where
       res.classId != res.knownClassId and
       res.knownClassId = types.id and
       types.kindOfType = c.value and
       (c.name like '%CLASS' or
              c.name like '%INTERFACE')
;


SELECT DISTINCT
       child.id AS child_ID,
       child.fullName AS child_FullName,
       parent.id AS parent_ID,
       parent.fullName AS parent_FullName,
       childFile.pathname AS child_FileName,
       parentFile.pathname AS parent_FileName
FROM
       TTypes child,
       TTypes parent,
       VKnowsRel vKnowsRel,
       TInheritances i,
       TSourceEntities childSource,
       TSourceEntities parentSource,
       TFiles childFile,
       TFiles parentFile
WHERE
       i.classId = child.id AND
       i.superId = parent.id AND
       vknowsRel.classId = parent.id AND
```

```
        vknowsRel.knownClassId = child.id AND
        /* auch statische Abhängigkeiten sollten berücksichtigt werden */
        /* vknowsRel.isStatic = false AND */
        childSource.id = child.id AND
        parentSource.id = parent.id AND
        childfile.id = childsource.sourcefileid AND
        parentfile.id = parentsource.sourcefileid
;


DROP TABLE VKnowsRel;
```

Lösungsstrategie

Die Vererbungsstruktur muss überarbeitet werden. Dabei muss unterschieden werden, wie die Abhängigkeit zu Stande kommt. Ein Spezialfall von Klasseninzest ist simulierte Polymorphie, die durch Unterklassenbildung aufgelöst werden kann. Eine weitere Möglichkeit ist die konsequente Verwendung von allgemeinen Schnittstellen in der Oberklasse.

2.1.21 Klässchen

Definition

Eine öffentlich sichtbare, nicht geschachtelte Klasse (Interfaces werden nicht betrachtet) ist sehr klein, d.h. sie bietet weder 3 eigene Methoden noch besitzt sie 3 eigene Attribute (inkl. Konstanten).

Beschreibung

Bei sehr kleinen Klassen stellt sich die Frage, ob diese Klasse wirklich sinnvoll ist, oder ob sie nicht besser mit einer anderen Klasse verschmolzen werden sollte. Oft entstehen Klässchen dadurch, dass zunächst Funktionalität dieser Klasse in andere Klassen verschoben wird, die Klasse selbst dann aber nicht entsprechend aufgelöst wird. Eine zu kleine Klasse stellt eine unnötige Abstraktion dar, die für das Verständnis des Systems eher hinderlich ist. Das Verständnis wird umso schwieriger, desto mehr Klässchen im System vorhanden sind.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Zählen von eigenen Attributen und Methoden für jede öffentlich sichtbare, nicht-geschachtelte Klasse (abstrakt oder konkret).
- Jede Klasse, die entweder weniger als drei Methoden oder drei Attributen/Konstanten hat, stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: Klaesschen.sql,v 1.3 2006/01/25 20:27:21 mihaylov Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:

Eine öffentlich sichtbare, nicht geschachtelte Klasse ist sehr klein,
d.h. sie bietet weder 3 eigene Methoden noch besitzt sie 3 eigene
Attribute (inkl. Konstanten).

Beschreibung:
```

```
Bei sehr kleinen Klassen stellt sich die Frage, ob diese Klasse wirklich
sinnvoll ist, oder ob sie nicht besser mit einer anderen Klasse verschmolzen
werden sollte. Oft entstehen Klässchen dadurch, dass zunächst Funktionalität
dieser Klasse in andere Klassen verschoben wird, die Klasse selbst dann aber
nicht entsprechend aufgelöst wird. Eine zu kleine Klasse stellt eine unnötige
Abstraktion dar, die für das Verständnis des Systems eher hinderlich ist. Das
Verständnis wird umso schwieriger, desto mehr Klässchen im System vorhanden
sind.
*/


create table  VClassMethods as

SELECT
       type.id as TypeId,
       type.fullName as TypeName,
       type.classid as classid,
       count(distinct func.id) as noContainedFunctions
FROM
       TTypes as type left outer join
       TFunctions as func on type.id = func.classId left outer join
       TSourceEntities sFunc on (sFunc.id = func.id) join
       TFiles fFunc on (sFunc.sourcefileid = fFunc.id) join
       TConstants cFile on (fFunc.kindOfFile = cFile.value) join
       TConstants as con on func.kindOfFunction = con.value

WHERE
       cFile.name = 'FILE_SOURCE' AND
              (con.name = 'FUNC_METHOD' OR
               con.name = 'FUNC_CONSTRUCTOR' OR
               con.name = 'FUNC_DESTRUCTOR')
GROUP BY
       type.id,
       type.fullName,
       type.classid
;

create index TClassMethodsIndex on VClassMethods (TypeId, TypeName, Classid);

create table VClassFields as

SELECT
       type.id as TypeId,
       type.fullName as TypeName,
       type.classid as classid,
       count(distinct field.id) as noContainedFields
FROM
       TTypes as type left outer join
       TVariables as field on type.id = field.classId join
       TConstants as con on field.kindOfVariable = con.value and
              (con.name like 'VAR_FIELD' or
              con.name like 'VAR_PROPERTY')
WHERE
       field.name <> '<self>'
GROUP BY
       type.id,
       type.fullName,
       type.classid
;

create index TClassFieldsIndex on VClassFields (TypeId, TypeName, Classid);


Select
       type.id,
       type.name,
       res2.TypeId as TypeId,
       res2.TypeName as TypeName,
       res2.noContainedFunctions,
       res3.noContainedFields,
       con.name
FROM
       TTypes as type join
       TMembers mType on (mType.id = type.id) left outer join
       TConstants con2 on (type.kindoftype = con2.value) left outer join
       TConstants con on (mType.visibility = con.value) left outer join
       VClassMethods as res2 on (type.id = res2.TypeID) left outer join
       VClassFields as res3 on type.id = res3.TypeID join
       TSourceEntities source on (type.id = source.id) join
       TFiles file on (source.sourcefileid = file.id)
```

```
WHERE
        con2.name = 'TYPE_CLASS' AND
        /* Keine inneren Klassen */
        type.classid = '-1' AND
        /* Nur öffentliche Klassen */
        con.name = 'VISIBILITY_PUBLIC'


EXCEPT

/* Die Klassen mit mehr als zwei Methoden oder Attributen ausschließen */

Select
        type.id,
        type.name,
        res2.TypeId as TypeId,
        res2.TypeName as TypeName,
        res2.noContainedFunctions,
        res3.noContainedFields,
        con.name
FROM
        TTypes as type join
        TMembers mType on (mType.id = type.id) left outer join
        TConstants con2 on (type.kindoftype = con2.value) left outer join
        TConstants con on (mType.visibility = con.value) left outer join
        VClassMethods as res2 on (type.id = res2.TypeID) left outer join
        VClassFields as res3 on type.id = res3.TypeID join
        TSourceEntities source on (type.id = source.id) join
        TFiles file on (source.sourcefileid = file.id)
WHERE
        con2.name = 'TYPE_CLASS' AND
        (res2.noContainedFunctions > 2 OR res3.noContainedFields > 2)
        AND

        /* Keine inneren Klassen */
        type.classid = '-1' AND
        /* Nur öffentliche Klassen */
        con.name = 'VISIBILITY_PUBLIC'


;

drop table VClassFields;
drop table VClassMethods;
```

Lösungsstrategie

Zunächst muss entschieden werden ob die Existenz des Klässchens (z.B. als Spezialisierung „tief“ in einer Vererbungshierarchie semantisch gerechtfertigt ist. Wenn das nicht der Fall ist, muss die Funktionalität der Klasse auf eine oder mehreren der anderen Klassen aufgeteilt werden. Dabei kann von den Methoden des Kläschens ausgegangen werden. Es wird nach Klassen gesucht, die eine starke Kopplung zum Beispiel durch Methodenaufrufe zum Klässchen aufweisen.
Beim Verschieben der Attribute bzw. beim Verschmelzen der Klasse mit einer anderen Klasse ist darauf zu achten, dass die Objekte zur Laufzeit in einer 1-1-Beziehung stehen. Andernfalls, müssen entsprechende Anpassungen zur Multiplizität der Assoziation in der Hostklasse vorgenommen werden.

2.1.22 Konstantenregen

Definition

Ein Konstantenregen liegt vor, wenn globale Konstanten (public static final bzw. public static const) namentlich mehrfach unabhängig vom Typ im System vorkommen.

Beschreibung

Globale Konstanten sind Konstanten, die für die Verwendung an vielen verschiedenen Stellen des Systems deklariert sind. Da in großen Systemen i.d.R. eine Vielzahl von Konstanten notwendig ist, bedarf es eines dedizierten Konstantenkonzeptes.
In jedem Fall sollte vermieden werden, dass global sichtbare Konstanten namentlich mehrfach vorkommen, da ansonsten unterschiedliche Inhalte mit demselben Namen angesprochen werden können. In jedem Fall besteht bei einem Konstantenregen die Gefahr, dass ein Wiederfinden, Anpassen und Ändern einer speziellen Konstante nur mit großem Zusatzaufwand möglich ist. Besitzen diese redundanten Konstanten unterschiedliche Werte ist vermutlich darüber hinaus die Fachlichkeit gefährdet. Eine Änderung eines Konstantenwertes muss dann u.U. an mehreren anderen Stellen nachgezogen werden. Wird eine Stelle vergessen, gerät das Verhalten des Programms eventuell außer Kontrolle.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermittlung aller global sichtbaren Konstanten (public static final bzw. public static const)
- Gruppierung der Konstanten mit gleichem Namen ohne Beachtung des Typs
- Jede Gruppe mit mindestens zwei Elementen (Konstanten) stellt eine Instanz des Problems dar.

Erkennung mit SISSy

```
/* $Id: Konstantenregen.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:

Ein Konstantenregen liegt vor, wenn globale Konstanten (public static
final bzw. public static const) in mehr als in einem zentralen
Konstanten-Paket deklariert sind.

Beschreibung:

Globale Konstanten sind Konstanten, die für die Verwendung an vielen
verschiedenen Stellen des Systems deklariert sind. Werden diese nicht
zentral an einem Ort abgelegt, besteht die Gefahr, dass ein Wiederfinden,
Anpassen und Wiederverwenden nur mit großem Zusatzaufwand möglich ist.
Darüber hinaus besteht in einem solchen Fall die Gefahr, dass globale
Konstanten mit identischem Namen mehrfach deklariert und gepflegt werden.
Besitzen diese redundanten Konstanten unterschiedliche Werte ist vermutlich
die Fachlichkeit gefährdet. Eine mehrfache Definition von Konstanten mit
gleichem Namen und gleichem Inhalt ist eine spezielle Form von
Codeduplikation, und als solche auf jeden Fall zu vermeiden. Eine Änderung
an einer Stelle muss an mehreren anderen Stellen nachgezogen werden. Wird
eine Stelle vergessen, gerät das Verhalten des Programms eventuell außer
Kontrolle.

Implementierung:

Diese Anfrage liefert eine Liste von Konstanten die von fremden Paketen
referenziert werden zusammen mit dem Anzahl von diesen Paketen.

*/

CREATE OR REPLACE VIEW VConstants AS

SELECT
        var.id as ConstantId,
        var.name as ConstantName,
        files.pathname
FROM
        TVariables var join
        TConstants cVar on (var.kindofvariable = cVar.value) join
```

```
        TMembers mVar on (mVar.id = var.id) join
        TConstants cMem on (cMem.value = mVar.visibility) join
        TSourceEntities sourceEntities on (var.classid = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id)
WHERE
        cVar.name = 'VAR_FIELD' AND
        var.isconst = 1 AND
        mVar.isstatic = 1 AND
        cMem.name = 'VISIBILITY_PUBLIC'
;

SELECT DISTINCT
        con1.ConstantName
FROM
        VConstants con1,
        VConstants con2
WHERE
        con1.ConstantName = con2.ConstantName AND
        con1.ConstantId <> con2.ConstantId AND
        con1.ConstantId < con2.ConstantId

;

DROP VIEW VConstants;
```

Lösungsstrategie

Eine Allgemeine Lösung kann nicht angegeben werden, da zunächst geklärt werden muss, ob die Konstanten wirklich für das ganze System, oder nur für einen bestimmten Systemteil gelten. Für die Systemkonstanten, bei denen die globale Sichtbarkeit tatsächlich notwendig ist, besteht die Lösung des Problems im Anlegen eines solchen Konstantenpools und dem Auslagern der globalen Konstanten in entsprechende Konstantenklassen bzw. - Interfaces innerhalb dieses Pools. Für Systemkonstanten mit einer übergroßen Sichtbarkeit muss die Konstante auf die maximal notwendige Sichtbarkeit heruntergesetzt werden.

2.1.23 Labyrinthmethode

Definition

Eine Labyrinth-Methode ist eine Methode mit beliebiger Sichtbarkeit und einem zyklomatischen Wert > 10. Unter dem zyklomatischen Wert wird hierbei die Anzahl linear unabhängiger Kontrollflusspfade innerhalb der Methode verstanden. Je mehr kontrollflussmodifizierende Anweisungen wie z.B. if, else, case die Methode enthält, desto höher ist ihr zyklomatischer Wert (vgl. [Mccabe76].

Beschreibung

Je mehr Verzweigungen der Kontrollfluss einer Methode enthält, desto schwieriger ist es, eine Methode zu verstehen, zu testen (z.B. nach dem C0 und C1-Kriterium, vgl. z.B. [PaSi94]) und weiterzuentwickeln. Methoden mit einer hohen Kontrollflusskomplexität sind zudem fehleranfälliger als andere Methoden, da sie aufgrund der hohen Anzahl an Kontrollflusspfaden nur schwer zu verstehen sind. Ebenso wird ein vollständiger Test der Methode immer aufwendiger und letztlich unwahrscheinlicher.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

- Ermittlung aller Methoden mit einer Zyklomatischen-Komplexität (McCabe-Zahl) >10.
- Jede so ermittelte Methode stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: LabyrinthMethode.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:

Eine Labyrinth-Methode (unabhängig Sichtbarkeit) hat einen zyklomatischen
Wert > 10. Unter dem zyklomatischen Wert wird hierbei die um 1 erhöhte
Anzahl Kontrollfluß modifizierender Anweisungen unabhängig der Schachtelung
betrachtet. Die kontrollflussmodifizierenden Anweisungen sind if, else,
cases und alle Schleifen.

Beschreibung:

Je mehr Verzweigungen der Kontrollfluss einer Methode enthält, desto
schwieriger ist es, eine Methode zu verstehen, zu testen (z.B. nach dem C0
und C1-Kriterium) und weiterzuentwickeln. Methoden mit einer hohen
Kontrollflusskomplexität sind zudem fehleranfälliger als andere Methoden,
da der Zustandsraum, der einen bestimmten Pfad durch die Methode bewirkt,
häufig unklar ist.

Implementierung:

Diese Anfrage liefert alle Funktionen mit einem zyclomatischen Wert hoher als
10.

*/

SELECT
       P.id as PackageId,
       P.fullName as PackageName,
       C.id as ContainingClassId,
       C.fullName as ContainingClassName,
       func.id as FunctionId,
       sig.signature as FunctionSignature,
       func.numberOfEdges - func.numberOfNodes + 1 as CyclomaticComplexity
FROM
       TFunctions func INNER JOIN
       TSignatures sig on func.id = sig.functionId LEFT OUTER JOIN
       TTypes C on func.classId = C.id LEFT OUTER JOIN
       TPackages P on C.PackageId = P.id OR
              func.packageId = P.id
WHERE
       func.numberOfEdges - func.numberOfNodes + 1 > 10 /* SCHWELLWERT */
ORDER BY
       CyclomaticComplexity DESC
;
```

Lösungsstrategie

Bei der Behebung von Labyrinthmethoden ist das manuelle Eingreifen unvermeidlich. In Abhängigkeit von der dahinterliegenden Ursache sind folgende alternative Strategien möglich:

- Wenn die Methode eine oder mehreren große bedingte Anweisungen (z.B. „switch“ oder geschachtelte „if-else“) hat, und wenn andere Methoden in der selben Klasse ähnliche Anweisungen enthalten, ist das ein Hinweis darauf dass Polymorphie angebracht ist. Das heißt, jeder Zweig in den komplexen bedingten Anweisungen eine Spezialisierung der Klasse darstellen könnte. In diesem Fall besteht die Lösung darin, dass diese alternativen Zweige in spezialisierenden Methoden von entsprechenden Unterklassen extrahiert werden.
- In der Mehrheit der Situationen besteht die Lösung darin, dass man aus der Methode neue Methoden (Helfermethoden) extrahiert („extract method“ Restrukturierung aus [Fowler00]). Eine nützliche und sehr effektive Heuristik, die man hier verwenden kann, ist, dass Methodenfragmente, die gesondert kommentiert sind, gute Kandidaten für Helfermethoden sind. Die Namen der neu-extrahierten Methoden können auf Basis dieser Kommentare festgelegt werden. Eine weitere meist effektive Maßnahme ist die Restrukturierung "Decompose Conditional". Hier wird für die Bedingung, den if- und den else-Teil jeweils eine eigene Methode angelegt. Wenn die Methoden adäquat benannt werden, steigt die Verständlichkeit der Methode enorm an.

Wenn die Methode besonders viele temporäre Variablen bzw. Parameter hat, und wenn das auch semantisch begründet ist, kann man ein so genanntes Methodenobjekt (eigentlich eine neue Klasse) extrahieren. Die

ursprüngliche Methode und die extrahierten Helfermethoden werden dabei in eine neue Klasse verschoben, und die alten Variablen und Methodenparameter werden Attribute der neuen Klasse.

2.1.24 Insiderwissen

Definition

Auf protected Member wird von außerhalb der Vererbungshierarchie zugegriffen.

Beschreibung

Der Zugriff auf protected Member (Methoden/Attribute) von außerhalb der Vererbungshierarchie ist nicht gestattet, da dies das Prinzip der Kapselung verletzt. Auf protected Attribute sollte nur mit Hilfe von Zugriffsmethoden zugegriffen werden. Für protected Methoden sollte überlegt werden, ob sie wirklich teil der öffentlichen Schnittstelle sein sollen.
Methoden, die für die Verwendung in Unit-Tests vorgesehen sind, müssen die Sichtbarkeit public oder package (bzw. default) haben

Programmiersprache

~~C++~~, Java, ~~C#~~, ~~Delphi~~

Betrachtungsebene

Designebene

Informale Erkennung

Gesucht werden

- protected Attribute, auf die von Klassen außerhalb der Vererbungshierarchie zugegriffen wird und
- protected Methoden, die von Methoden außerhalb der Vererbungshierarchie aufgerufen werden.

Erkennung mit SISSy

```
/* $Id: Insiderwissen.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Beschreibung:
 Eine Klasse A bietet Attribute öffentlich an, worauf von ausserhalb der Klasse A und deren
Vererbungshierarchie zugegriffen wird.

@TODO: Momentan werden nur Zugriffe auf Attribute überprüft.
@TODO: Anfrage überdenken
*/

SELECT
       A.id AS ContainingClassId,
       A.fullName AS ContainingClassFullName,
       field.id AS AccessedFieldId,
       field.name AS AccessedFieldName,
       B.id AS AccessingClassId,
       B.fullName AS AccessingClassFullName
FROM
       TTypes A,
       TMembers memberAfield,
       TVariables AS field,
       TConstants constantField,
       TConstants constantPublic,
       TAccesses acc,
       TTypes B
WHERE
       memberAfield.ClassId = A.Id AND
       memberAfield.Id = field.Id AND
```

```
        field.KindOfVariable = constantField.Value AND
        constantField.Name = 'VAR_FIELD' AND
        memberAfield.visibility = constantPublic.Value AND
        constantPublic.Name = 'VISIBILITY_PUBLIC' AND
        NOT (memberAfield.isStatic = 1 AND
                memberAfield.isFinal = 1) AND
        acc.targetId = field.id AND
        acc.classId = B.id AND

        B.id != A.id AND

        /* B ist nicht in der Vererbungshierarchie von A enthalten */
        NOT EXISTS (
                SELECT *
                FROM
                        TInheritances inheritanceBA
                WHERE
                        inheritanceBA.superId = A.id AND
                        inheritanceBA.classId = B.id
        )
;
```

Lösungsstrategie

Als erstes muss überlegt werden ob das Attribut oder die Methode überhaupt in der Klasse gehört oder es um ein deplaziertes Attribut /Methode handelt. Nachdem das Attribut korrekt platziert wurde, muss eine Sichtbarkeit (am liebsten private Sichtbarkeit) für das Attribut ausgewählt werden. Ziel soll sein, das Attribut hinter höherwertigen Services zu „verstecken“. Die einfache Ersetzung der Direktzugriffe durch Getter- und Setter-Methoden ist zwar besser als nichts, aber in der Regel keine vollständige Lösung.
Eine vollständige Lösung des Problems für Attribute kann nur als Resultat einer Analyse der Schnittstelle der Klasse selbst und der Klienten des Attributs erfolgen, gefolgt von einer Restrukturierung der Klasse der und Klienten, so dass zumindest Zustandsänderungen in der Klasse möglichst durch höherwertige Services und nicht direkt über setter Methoden erfolgt. Im Fall von Methoden die als „protected“ deklariert sind aber außerhalb des Vererbungsbaums referenziert werden (aber aus dem selben Paket), muss eine Analyse durch den Entwickler durchgeführt werden. Eventuell ist eine Änderung der Sichtbarkeit dieser Methoden auf „public“ gerechtfertigt.

2.1.25 Lange Parameterliste

Definition

Eine lange Parameterliste liegt vor, wenn eine Methode (inkl. Struktoren) mehr als 7 Übergabe-Parameter besitzt. Optionale Parameter werden hierbei mit betrachtet, variable Parameterlisten zählen als einzelner Parameter. Nicht betrachtet werden Methoden, die aus externen Bibliotheken stammen und überschrieben bzw. implementiert werden.

Beschreibung

Methoden/Struktoren mit sehr vielen Parametern sind nur schwer zu verwenden, zu verstehen und sehr änderungsanfällig. Während in nicht OO-Sprachen die einzige Alternative zu Methodenparametern globale Daten sind, steht mit Objekten heute die Möglichkeit zur Verfügung, dass Methoden sich die nötigen Informationen über ein übergebenes Objekt selbst beschaffen.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Design

Informale Erkennung

- Ermittle alle Methoden und Struktoren mit mehr als 7 Parametern.
- Ignoriere diejenigen Methoden, die Methoden aus externen Bibliotheken überbeschreiben oder implementieren.
- Jede verbleibende Methode bzw. jeder Struktor stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: LangeParameterliste.sql,v 1.3 2006/02/09 16:54:33 seng Exp $ */
/* FZI Forschungszentrum Informatik */


SELECT
	P.id as PackageId,
	P.fullName as PackageName,
	C.id as ContainingClassId,
	C.name as ContainingClassName,
	func.functionId as FunctionId,
	func.signature as FunctionSignature,
	cnt as NumberOfParameters
FROM
	TSignatures func
INNER JOIN
	TModelElements elem on func.functionId = elem.id
JOIN
	TConstants cElem on (cElem.value = elem.status and cElem.name = 'STATUS_NORMAL')
JOIN
	TMembers member on (func.functionId = member.id)
LEFT OUTER JOIN
	TTypes C on elem.ParentId = C.id
LEFT OUTER JOIN
	TPackages P on
		C.PackageId = P.id
		OR elem.parentId = P.id
INNER JOIN
	(SELECT DISTINCT
		COUNT(parameter) AS cnt,
		f.id as fId
	FROM
		TVariables parameter,
		TConstants constantFormalParameter,
		TFunctions f
	WHERE
		parameter.KindOfVariable = constantFormalParameter.Value AND
		constantFormalParameter.Name = 'VAR_FORMALPARAM' AND
		f.id = parameter.functionId
	GROUP BY
		fId
	) as res on func.functionId = res.fId
WHERE
	cnt > 7
		AND not exists (
		SELECT
			*
		FROM
			TMembers overriddenmember,
			TModelElements mElement,
			TConstants mCon
		WHERE
			    member.overridenmemberid = overriddenmember.id
			AND mElement.id = overriddenmember.id
			AND mCon.value = mElement.status
			AND mCon.name = 'STATUS_LIBRARY'
		)


ORDER BY
	cnt DESC
;
```

Lösungsstrategie

Zur Entfernung von langen Parameterlisten gibt es drei Standardrefactorings:

- "Replace Parameter with Method" Anstatt einen Parameter zu übergeben, wird das benötigte Objekt durch einen Methodenaufruf auf einem Feld oder einem anderen Parameter besorgt.
- "Preserve whole object" Anstatt ein Feld einer Klasse als Parameter zu übergeben, wird die komplette Klasse übergeben.
- "Introduce Parameter Object" Parameter, die immer wieder zusammen verwendet werden, können zu einer eigenen Klasse zusammengefasst werden.

Eventuell kann die Funktion in zwei Funktionen aufgeteilt werden.

2.1.26 Maskierende Datei

Definition

Eine maskierende Datei liegt vor, wenn der Name der Datei (ohne Erweiterung; bei C++ sowohl Header- als auch Implementierungsdatei) in keinem Namen einer der enthaltenen, nicht-geschachtelten Klassen mit der größten Sichtbarkeit vollständig und in identischer Schreibweise (inkl. Groß- und Kleinschreibung) enthalten ist.

Beschreibung

In den betrachteten Programmiersprachen wird der Code immer in Dateien abgelegt. Während in JAVA zwischen dem Inhalt und der umgebenden Datei eine namentliche 1:1-Abbildung bei öffentlichen Klassen erzwungen wird, gilt dies für C++ und in Java für package-private Klassen nicht. Die grundsätzlich gegebene Möglichkeit, mit der Benamung jeglichen Zusammenhang zwischen enthaltener Klasse und Dateinamen zu verdecken, erschwert deutlich das Wiederfinden von Klassen. Der Name einer Datei sollte daher in jedem Fall in dem Namen mindestens einer enthaltenen Klasse enthalten sein. Mögliche Namenserweiterungen in Form von Präfixen (z.B. "C") oder Postfixen (z.B. "Impl") erschweren das Wiederfinden nicht.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Technologieebene

Informale Erkennung

- Ermitteln der Namen der nicht-geschachtelten Klassen mit der höchsten Sichtbarkeit innerhalb jeder Datei.
- Überprüfen, ob in mindestens einem ermittelten Klassennamen der Dateiname vollständig enthalten ist.
- Alle Dateien bei denen diese Forderung nicht erfüllt ist, stellen eine Instanz des Problems dar.

Erkennung mit SISSy

```
/* $Id: MaskierendeDatei.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:
```

```
Eine maskierende Datei liegt vor, wenn der Name der Datei (bei C++
sowohl Header- als auch Implementierungsdatei) keinen Namen einer
der enthaltenen, nicht-geschachtelten Klassen mit der größten
Sichtbarkeit vollständig und in identischer Schreibweise (inkl.
Groß- und Kleinschreibung) enthält.

Beschreibung:

In den betrachteten Programmiersprachen wird der Code immer in
Dateien abgelegt. Während in JAVA zwischen dem Inhalt und der
umgebenden Datei eine namentliche 1:1-Abbildung erzwungen wird,
gilt dies für C++, C# und Delphi nicht. Die grundsätzlich gegebene
Möglichkeit, mit der Benamung jeglichen Zusammenhang zwischen
enthaltener Klasse und Dateinamen zu verdecken, erschwert deutlich
das Wiederfinden von Klassen. Der Name einer Datei sollte daher in
jedem Fall den Namen mindestens einer enthaltenen Klasse enthalten.
Mögliche Namenserweiterungen in Form von Präfixen (z.B. "C") oder
Suffixen (z.B. "Impl") erschweren das Wiederfinden nicht.

Implementierung:

Diese Implementierung liefert alle Dateinamen und Klassennamen die nicht
gleich oder erweitert sind. Siehe oben.

*/

create or replace view VShortFileNames as

select
       f.id as FileId,
       substring(substring(f.pathName  from  '[\\\\/][a-zA-Z0-9\\_]*\\.[a-zA-Z0-9\\_]*$')  from
2 for
              strpos(substring(f.pathName  from  '[\\\\/][a-zA-Z0-9\\_]*\\.[a-zA-Z0-9\\_]*$'),
'.') - 2) as ShortFileName
from
       TFiles f
;


select  Distinct
       f.id as SourceFileID,
       f.pathname as SourceFileName
from
       TFiles f join
       TConstants con on (f.kindoffile = con.value)
where
       con.name = 'FILE_SOURCE'


except

select  distinct
       f.id as SourceFileId,
       f.pathname as SourceFileName

from
       TFiles f join
       VShortFileNames v on v.fileId = f.id join
       TSourceEntities s on f.id = s.sourceFileId join
       TTypes t on t.id = s.id join
       TMembers m on t.id = m.id and
              m.classId = -1
where
       strpos(t.name, v.shortFileName) <> 0
;



drop view VShortFileNames;
```

Lösungsstrategie

Die Behebung einer maskierten Klasse geschieht durch das Umbenennen der zugehörigen Datei oder in dem Fall, dass eine überbuchte Datei vorliegt, wie dort beschrieben.

2.1.27 Nachlässige Kommentierung

Definition

Ein gut kommentiertes Software-System besitzt als notwendiges Kriterium einen möglichst großen Quotient aus Kommentarzeilen und Brutto-Lines-Of-Code. Auf der anderen Seite ist ein überkommentiertes Software-System mit deutlich mehr Kommentar- als Programmzeilen ebenfalls suboptimal, da der Code in der Kommentierung untergeht. Als ideal kommentiertes System wird daher ein System verstanden, bei dem jede Programmcodezeile eine Kommentarzeile besitzt. Für eine einheitliche Interpretation dieses Qualitätsindikators „nachlässige Kommentierung“ bedeutet dies, dass

*Abs(BLOC-2*Commentlines)*

möglichst klein sein muß.
Die konkreten QBL-Grenzwerte ergeben sich aus den Quartilen des QBL-Repositories. Als Kommentarzeilen werden sowohl alle reinen Zeilen von Blockkommentaren (häufig eingeleitet durch `/*`) als auch reine strategische Kommentare (häufig eingeleitet durch `//`) betrachtet. Kommentare hinter Programmcode werden nicht berücksichtigt.

Beschreibung

Dieses Problemmuster adressiert den systemweiten Kommentierungsgrad als notwendiges Kriterium die Wartungsphase unterstützender Dokumentation. Liegt dieser Wert deutlich oberhalb dessen, was im QBL-Repository abgelegt ist, so befinden sich systemweit deutlich weniger Kommentare als üblich. Unabhängig vom Inhalt der wenigen Kommentare kann daher vermutet werden, daß die Kommentierung ungenügend ist und daher jegliche Arbeit am Code deutlichen Mehraufwand bedeutet und ggf. fehlerträchtig ist.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

- Zählen der absolute Anzahl an Kommentarzeilen (CLOC)
- Zählen der insgesamt vorhandenen Zeilen (BLOC)
- Berechnen der Formel abs(BLOC-2*CLOC)

Erkennung mit SISSy

```
/* $Id: NachlaessigeKommentierung.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:

Der Quotient aus Kommentarzeilen und Brutto-Lines-Of-Code sollte möglichst
hoch sein. Die konkreten Grenzwerte ergeben sich aus den Quartilen des
QBL-Repositories (vgl. unten). Als Kommentarzeilen werden sowohl alle
Zeilen von Blockkommentaren (häufig eingeleitet durch "xx") als auch
strategische Kommentare (häufig eingeleitet durch "//" und ebenfalls zusammen
mit Code in einer Zeile möglich) betrachtet.

Beschreibung:

Dieses Problemmuster adressiert den systemweiten Kommentierungsgrad als
notwendiges Kriterium die Wartungsphase unterstützender Dokumentation.
Liegt dieser Wert deutlich unterhalb dessen, was im QBL-Repository abgelegt
ist, so befinden sich systemweit deutlich weniger Kommentare im System als
üblich. Unabhängig vom Inhalt der wenigen Kommentare kann daher vermutet
werden, daß die Kommentierung ungenügend ist und daher jegliche Arbeit am
Code deutlichen Mehraufwand bedeutet und ggfs. fehlerträchtig ist.

Implementierung:
```

```
In dieser Implementierung wird das Verhaeltnis von Kommentarzeilen zu "purem"
Quellcode (inkl. Leerzeilen) als Basis zur Berechnung vom "Mass der
Kommentierung" verwendet. Das Verhaeltnis wird pro Klasse untersucht.

(OS) Berechnung an Beschreibung angepasst. Damit die Werte konsistent sind,
berechnen wir 1 - Verhaeltnis als Ergebnis.

*/



CREATE or replace VIEW Vcomments as
SELECT
        sum(comments.numberoflines) as CLOC
FROM
        TComments comments
;


CREATE or replace VIEW Vlinesofcode as

SELECT
        sum(files.linesofcode) as LOC
FROM
        TFiles files

;

SELECT
        abs(linesofcode.LOC - 2 * comments.CLOC) as ratio,
        comments.CLOC,
        linesofcode.LOC
FROM
        Vcomments comments,
        Vlinesofcode linesofcode
;

DROP View Vcomments;
DROP View Vlinesofcode;
```

Lösungsstrategie

Zu wenig Kommentierung erschwert die Wartung. Aber auch zu viel Kommentierung kann genauso, oder noch schädlicher sein als zu wenig Kommentierung (siehe auch [Fowler00]). Gute Kommentare sollten übermitteln was ein Stück Code macht und eventuell warum, aber nicht wie, denn anders ziehen alle Änderungen im kommentierten Codeblock, Änderungen im Kommentar mit sich. Gute Kommentare zu schreiben erfordert eine gewisse persönliche Einstellung des Programmierers und des Projektmanagements.

2.1.28 Namensfehler

Definition

Für bestimmte Artefakte der Programmierung sind rudimentäre Namenskonventionen gefordert:

- Ein Paketname entspricht dem regulären Ausdruck [a-zA-Z][a-zA-Z0-9_]*
- Ein Klassenname/Interfacename (Ausnahme: anonyme Klassen) entspricht dem regulären Ausdruck [A-Z][a-zA-Z0-9_]*
- Ein Dateiname entspricht dem regulären Ausdruck [a-zA-Z][a-zA-Z0-9_]*\.[a-zA-Z]+
- Ein Methodenname (Ausnahme: Konstruktoren/Destruktoren) entspricht dem regulären Ausdruck [a-z][a-zA-Z0-9_]*
- Ein Konstantenname entspricht dem regulären Ausdruck [A-Z][A-Z0-9_]*
- Ein Attributname entspricht dem regulären Ausdruck [a-z][a-zA-Z0-9_]*

Wichtig ist an dieser Stelle vor allen Dingen die Konsistenz. Wird für ein Artefakttyp eine andere, sinnvolle Vorgabe zur Unterscheidung der oben genannten Artefakttypen verwendet, so kann diese Regel entsprechend dieser projektspezifischen Setzung umdefiniert werden. Diese Parametrisierbarkeit der Regel ist innerhalb des Code-Sünden-Index einmalig, betrachtet wird in jedem Fall mehr die Homogenität als die konkrete

Namensvorgabe. In C++ beispielsweise gibt es aber oft die Konvention, dass Methodennamen mit Großbuchstaben beginnen sollen.

Beschreibung

Namenskonventionen sind einfach zu etablierende "Spielregeln", die helfen, eine einheitliche Erwartungshaltung gegenüber den Artefakten aufzubauen. Einige Regeln helfen auch, zeichensatzspezifische Probleme in der Wartung (z.B. bei der Verwendung von ö, ä und ß) im Vorfeld auszuschließen.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Jede der oben aufgeführten Artefaktklassen wird separat nach Verletzungen der jeweilsgeltenden Namesschemata untersucht.
- Jedes Artefakt, welches das geltende Schema verletzt, stellt eine Instanz des Problems dar

Erkennung mit SISSy

```
/* $Id: Namensfehler.sql,v 1.2 2006/01/19 14:25:20 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:

Für bestimmte Artefakte der Programmierung sind rudimentäre Namenskonventionen
gefordert:
 - Ein Paketname entspricht dem regulären Ausdruck [a-zA-Z][a-zA-Z0-9_]*
 - Ein Klassenname/Interfacename (Ausnahme: anonyme Klassen) entspricht dem
 regulären Ausdruck [A-Z][a-zA-Z0-9_]*
 - Ein Dateiname entspricht dem regulären Ausdruck [a-zA-Z][a-zA-Z0-9_]*\.[a-zA-Z]+
 - Ein Methodenname (Ausnahme: Konstruktoren/Destruktoren) entspricht dem regulären
 Ausdruck [a-z][a-zA-Z0-9_]*
 - Ein Konstantenname entspricht dem regulären Ausdruck [A-Z][A-Z0-9_]*
 - Ein Attributname entspricht dem regulären Ausdruck [a-z][a-zA-Z0-9_]*

Beschreibung:

Namenskonventionen sind einfach zu etablierende "Spielregeln", die helfen, eine
einheitliche Erwartungshaltung gegenüber den Artefakten aufzubauen. Einige Regeln
helfen auch, zeichensatzspezifische Probleme in der Wartung (z.B. bei der Verwendung
von ö, ä und ß) im Vorfeld auszuschließen.

Implementierung:

Diese Anfrage liefert alle Modelelemente die haben Namensfehler.

*/

/* Pakete (nur solche die in keinem Assembly enthalten sind) */
SELECT DISTINCT
          p.id
        , 'PACKAGE' AS type
        , p.name AS name
        , p.fullname AS longname
FROM
        TPackages p
        LEFT JOIN
                TTypes t
        ON
                t.packageid = p.id
        LEFT JOIN
                TSourceEntities se
        ON
                se.id = t.id
                AND se.assemblyfileid <> -1
```

```
WHERE
           p.name NOT SIMILAR TO '[a-zA-Z][a-zA-Z0-9\\_]*'
       AND t.id IS NOT NULL
       AND se.id IS NULL

UNION

/* Klassen und Interface */
SELECT DISTINCT
         c.id
       , 'CLASS' AS type
       , c.name
       , p.fullname || '.' || c.name AS longname
FROM
         TTypes c
       , TPackages p
       , TConstants tc
       , TSourceEntities se
WHERE
           c.packageid = p.id
       AND c.name NOT SIMILAR TO '[A-Z][a-zA-Z0-9\\_]*'
       AND c.name <> '<unknownClassType>'
       AND c.name <> ''
       AND c.kindOfType = tc.value
       AND (
                tc.name LIKE 'TYPE_%CLASS'
           OR tc.name LIKE 'TYPE_%DELEGATE'
           OR tc.name LIKE 'TYPE_%INTERFACE'
       )
       AND se.id = c.id
       AND se.sourcefileid <> -1

UNION

/* Methoden */
SELECT DISTINCT
         m.id
       , 'METHOD' AS type
       , m.name
       , p.fullname || '.' || c.name || '@' || m.name AS longname
FROM
         TMembers m
       , TTypes c
       , TPackages p
       , TSourceEntities se
       , TConstants tc
       , TSignatures sig
WHERE
           c.packageId = p.id
       AND m.classId = c.id
       AND m.name <> '<unknownMethod>'
       AND m.name NOT SIMILAR TO '[a-z][a-zA-Z0-9\\_]*'
       AND m.kindOfMember = tc.value
       AND (
                tc.name LIKE 'FUNC_METHOD'
           OR tc.name LIKE 'FUNC_GENERIC'
       )
       AND sig.functionId = m.id
       AND se.id = c.id
       AND se.sourceFileId <> -1

       AND not exists (
              SELECT
                     *
              FROM
                     TMembers overriddenmember,
                     TModelElements mElement,
                     TConstants mCon
              WHERE
                         m.overridenmemberid = overriddenmember.id
                     AND mElement.id = overriddenmember.id
                     AND mCon.value = mElement.status
                     AND mCon.name = 'STATUS_LIBRARY'
              )


UNION

/* Attribute und Konstanten*/
SELECT DISTINCT
```

```
          a.id
        , 'FIELD' AS type
        , a.name
        , p.fullname || '.' || c.name || '@' || a.name AS longname
FROM
          TVariables a
        , TTypes c
        , TPackages p
        , TSourceEntities se
        , TConstants tc
WHERE
            c.packageId = p.id
        AND a.classId = c.id
        AND a.name <> '<self>'
        AND (
                a.isConst = 0 AND a.name NOT SIMILAR TO '[a-z][a-zA-Z0-9\\_]*'
            OR a.isConst = 1 AND a.name NOT SIMILAR TO '[A-Z][A-Z0-9\\_]*'
        )
        AND a.kindOfVariable = tc.value
        AND tc.name LIKE 'VAR_FIELD'
        AND se.id = c.id
        AND se.sourceFileId <> -1

UNION

/* Dateien */
SELECT DISTINCT
          f.id
        , 'FILE' AS type
        , f.pathName AS name
        , f.pathName AS longname
FROM
          TFiles f
        , TConstants tc
WHERE
            tc.value = f.kindOfFile
        AND tc.name = 'FILE_SOURCE'
        AND f.pathName NOT SIMILAR TO '%[\\\\/][a-zA-Z][a-zA-Z0-9\\_]*\\.[a-zA-Z]+'
;
```

Lösungsstrategie

Bezeichner, die sich nicht an die Konvention halten, müssen umbenannt werden.

2.1.29 Objektplacebo (Attribut)

Definition

Ein statisches Attribut (inkl. Konstanten) wird über eine Objekt-Instanz angesprochen.

Beschreibung

Falls ein statisches Attribut über ein Objekt angesprochen wird, suggeriert dies, dass es um ein Attribut handelt, auf das nur mittels eines Objekts zugegriffen werden kann. Insbesondere kann dies dazu führen, dass das Objekt nur zum Zugriff auf die statischen Merkmale erzeugt wird, und somit eigentlich überflüssig ist. Korrekt wäre es, das Attribut direkt über die Klasse anzusprechen.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermitteln aller Zugriffstellen auf statische Attribute (inkl. Konstanten).
- Prüfen, ob der Zugriff über ein Objekt oder eine Klasse erfolgt. Im ersteren Fall ist ein Objektplacebo gefunden.
- Jeder derartige Zugriff wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: ObjektPlaceboAttribut.sql,v 1.2 2006/01/24 15:26:16 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:

Ein statisches Attribut (inkl. Konstanten) wird über eine Objekt-Instanz
angesprochen.

Beschreibung:

Falls ein statisches Attribut über ein Objekt angesprochen wird, suggeriert dies,
dass es um ein Attribut handelt, auf das nur mittels eines Objekts zugegriffen
werden kann. Insbesondere kann dies dazu führen, dass das Objekt nur zum Zugriff
auf die statischen Merkmale erzeugt wird, und somit eigentlich überflüssig ist.
Korrekt wäre es, das Attribut direkt über die Klasse anzusprechen.

Implementierung:

Diese Anfrage liefert alle Zugriffspunkte, wo ein statisches Attribut über eine
Objekt-Instanz zugegriffen wird. Zugriffe zu derselben Klasse oder derselben
Klassenhierarchie gehörenden statischen Attibuten sind ignoriert.

*/

SELECT
        pack.id as SourcePackageId,
        pack.fullName as SourcePackageName,
        types2.id as SourceClassId,
        types2.fullName as SourceClassName,
        sig2.functionId as SourceFunctionId,
        sig2.signature as SourceFunctionSignature,
        types1.id as TargetClassid,
        types1.fullName as TargetClassName,
        fld.id as TargetFieldId,
        fld.name as TargetFieldName
FROM
        TAccesses accesses1 join
        TMembers members on (members.id = accesses1.targetid and
                members.isstatic = 1 ) join
        TTypes tt on tt.id = accesses1.classId join
        TVariables fld on members.id = fld.id join
        TAccesses accesses2 on (accesses1.sourceid = accesses2.sourceid and
                accesses2.position = accesses1.position - 1) join
        TConstants constants2 on (accesses2.kindofaccess = constants2.value and
                constants2.name != 'TYPEACCESS_STATIC') join
        TTypes types1 on members.classId = types1.id left outer join
        TTypes types2 on accesses1.classId = types2.id left outer join
        TSignatures sig2 on accesses1.functionId = sig2.functionId left outer join
        TFunctions f on sig2.functionid = f.id left outer join
        TPackages pack on f.packageId = pack.id
ORDER BY
        pack.id,
        types2.id,
        sig2.functionId,
        types1.id,
        fld.id
;
```

Lösungsstrategie

Für jeden Zugriff auf ein Klassenattribut muss zunächst die Klasse bestimmt werden, in der das Klassenattribut deklariert ist. Dann müssen die Zugriffe, die über ein Objekt erfolgen, durch Zugriffe über die Klasse ausgetauscht werden.

2.1.30 Objektplacebo (Methode)

Definition

Eine statische Methode wird über eine Objekt-Instanz angesprochen.

Beschreibung

Falls eine statische Methode über ein Objekt angesprochen wird, suggeriert dies, dass es sich hierbei um eine Methode handelt, auf die nur mittels eines Objekts zugegriffen werden kann und die daher auch die objekteigenen Attribute verwendet. Insbesondere kann dies dazu führen, dass das Objekt nur zum Zugriff auf die statischen Merkmale erzeugt wird, und somit eigentlich überflüssig ist. Korrekt wäre es, die Methode direkt über die Klasse anzusprechen.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermitteln aller Aufrufstellen von statischen Methoden.
- Prüfen, ob der Aufruf über ein Objekt oder ein Klasse erfolgt. Im ersteren Fall ist ein Objektplacebo gefunden.
- Jeder derartige Zugriff wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: ObjektPlaceboMethode.sql,v 1.2 2006/01/24 15:26:16 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:

Eine statische Methode wird über eine Objekt-Instanz angesprochen.

Beschreibung:

Falls eine statische Methode über ein Objekt angesprochen wird, suggeriert dies,
dass es sich hierbei um eine Methode handelt, auf die nur mittels eines Objekts
zugegriffen werden kann. Insbesondere kann dies dazu führen, dass das Objekt
nur zum Zugriff auf die statischen Merkmale erzeugt wird, und somit eigentlich
überflüssig ist. Korrekt wäre es, die Methode direkt über die Klasse anzusprechen.

Implementierung:

Diese Anfrage liefert alle Aufrufpunkte, wo eine statische Methode über eine
Objekt-Instanz aufgerufen wird. Aufrufe zu derselben Klasse oder derselben
Klassenhierarchie gehörenden statischen Methoden sind ignoriert.

*/

SELECT
        pack.id as SourcePackageId,
        pack.fullName as SourcePackageName,
        types2.id as SourceClassId,
        types2.fullName as SourceClassName,
        sig2.functionId as SourceFunctionId,
        sig2.signature as SourceFunctionSignature,
        types1.id as TargetClassid,
        types1.fullName as TargetClassName,
        sig.functionId as TargetMethodId,
        sig.signature as TargetMethodSignature,
        accesses2.position as Position
FROM
        TAccesses accesses1 join
```

```
        TMembers members on (members.id = accesses1.targetid and
                members.isstatic = 1 ) join
        TConstants con1 on (members.kindofmember = con1.value and
                con1.name = 'FUNC_METHOD') join
        TTypes tt on tt.id = accesses1.classId join
        TSignatures sig on members.id = sig.functionId join
        TAccesses accesses2 on (accesses1.sourceid = accesses2.sourceid and
                accesses2.position = accesses1.position - 1) join
        TConstants constants2 on (accesses2.kindofaccess = constants2.value and
                constants2.name != 'TYPEACCESS_STATIC') join
        TTypes types1 on members.classId = types1.id left outer join
        TTypes types2 on accesses1.classId = types2.id left outer join
        TSignatures sig2 on accesses1.functionId = sig2.functionId left outer join
        TFunctions f on sig2.functionid = f.id left outer join
        TPackages pack on f.packageId = pack.id
ORDER BY
        pack.id,
        types2.id,
        sig2.functionId,
        types1.id,
        sig.functionId
;
```

Lösungsstrategie

Für jeden Zugriff auf eine Klassenmethode muss zunächst die Klasse bestimmt werden, in der die Klassenmethode deklariert ist. Dann müssen die Zugriffe, die über ein Objekt erfolgen, durch Zugriffe über die Klasse ausgetauscht werden.

2.1.31 Paketchen

Definition

Ein nicht-leeres Paket besitzt weniger als 3 öffentliche Klassen bzw. Interfaces.

Beschreibung

Ein sehr kleines Paket bietet keine oder nur sehr wenige Funktionalität an. Somit scheint der Sinn des Paketes fraglich. Da ein überflüssiges Paket aber das Verständnis des Gesamtsystems erschwert, sollte die dort enthaltene Funktionalität besser auf andere Pakete verteilt werden.

Programmiersprache

C++, Java, C#, ~~Delphi~~

Betrachtungsebene

Architekturebene

Informale Erkennung

- Ermittlung der Pakete, die in der Summe weniger als 3 öffentliche Klassen und Schnittstellen (Interfaces) haben.
- Jedes derartige Paketchen stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: Paketchen.sql,v 1.1 2006/01/18 15:25:32 seng Exp $ */
/* FZI Forschungszentrum Informatik */



create or replace view VPackPublicTypes as

SELECT
        pack.id as PackageId,
        pack.fullName as PackageName,
```

```
        count(distinct types.id) as NoPublicTypes
FROM
        TPackages as pack left outer join
        TTypes as types on types.packageId = pack.id and types.classid = -1 join
        TMembers as m on types.id = m.id join
        TConstants con on m.Visibility = con.Value and con.Name = 'VISIBILITY_PUBLIC' join
        TSourceEntities as source on (source.id = types.id) join
        TFiles files on (source.sourcefileid = files.id)
GROUP BY
        pack.id,
        pack.fullName
;


SELECT
        t.PackageId,
        t.PackageName,
        t.NoPublicTypes
FROM
        VPackPublicTypes as t
WHERE
        t.NoPublicTypes < 3
        AND t.NoPublicTypes > 0
ORDER BY
        NoPublicTypes
;


drop view VPackPublicTypes;
```

Lösungsstrategie

Zuerst muss herausgefunden werden, was mit den Klassen im Paket geschehen soll. Entweder können alle mit einem anderen Paket verschmolzen werden, oder sie werden auf verschiedene Pakete verteilt. Wenn man zwei Pakete verschmilzt, muss man eventuell einen neuen Namen für das Paket vergeben. Danach kann das alte Paket gelöscht werden. In einem letzten Schritt kann überprüft werden, ob die Sichtbarkeit von Methoden und Attributen eingeschränkt werden kann, da manche Klassen jetzt nicht mehr über Paketgrenzen kommunizieren müssen.

2.1.32 Pakethierarchieaufbruch

Definition

Eine Klasse UK in einem Oberpaket OP erbt direkt von einer Klasse OK in einem direktem oder indirekten Unterpaket UP von OP. Vererbung schließt hier Interface-Implementierung mit ein.

Beschreibung

Die durch die Einteilung in Pakete vorgegebene Hierarchie suggeriert, dass Vererbungsbeziehungen nur zwischen Paketen auf gleicher Ebene oder von einem Paket einer niedrigeren Ebene zu einem Paket auf höher Ebene verlaufen sollen. Es ist nicht damit zu rechnen, dass eine Klasse UK aus einer Oberpaket OP von einer Klasse OK aus einem Unterpaket UP erbt, so dass die Verständlichkeit erschwert ist, da hier zwei Spezialisierungsrichtungen orthogonal zueinander stehen. Eventuell ist eine solche Vererbungsbeziehung ein Zeichen für eine schlechte Pakethierarchie.

Programmiersprache

C++, Java, C#, ~~Delphi~~

Betrachtungsebene

Architekturebene

Informale Erkennung

- Bilden von Oberklasse-Unterklasse Paaren, wobei die Paarbildung auf direkte Vererbungsbeziehungen beschränkt ist. Zu den Oberklassen zählen in diesem Fall auch Interfaces bzw. Implements-Beziehungen.
- Für jedes Paar prüfen, ob die Oberklasse aus einem direktem oder indirektem Unterpaket des Unterklassenpakets stammt.
- Jedes Paar, auf das dieses zutrifft, stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: PakethierarchieAufbruch.sql,v 1.1 2006/01/18 15:25:32 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*

Definition:

Eine Klasse K in einem Oberpaket OP erbt direkt von einer Klasse OK
in einem direktem oder indirekten Unterpaket UP von OP. Vererbung
schließt hier Interface-Implementierung mit ein.

Beschreibung:

Die durch die Einteilung in Pakete vorgegebene Hierarchie suggeriert,
dass Vererbungsbeziehungen nur zwischen Paketen auf gleicher Ebene
oder von einem Paket einer niedrigeren Ebene zu einem Paket auf höher
Ebene verlaufen sollen. Es ist nicht damit zu rechnen, dass ein Paket
aus einer Oberklasse von einer Klasse aus einem Unterpaket erbt, so
dass die Verständlichkeit erschwert ist. Eventuell ist eine solche
Verebungsbeziehung ein Zeichen für eine schlechte Pakethierarchie.

Implementierung:

Die Anfrage liefert nicht nur direkte Unterklassen als auch indirekte
Unterklassen und für jede solche Unterklasse zeigt die Vererbungstiefe.

(OS) Library-Pakete werden nicht mehr berücksichtigt

*/

SELECT
        baseclasspackage.id as BaseClassPackageId,
        baseclasspackage.fullname as BaseClassPackageName,
        baseclass.id as BaseClassId,
        baseclass.name as BaseClassName,
        subclasspackage.id as SubClassPackageId,
        subclasspackage.fullname as SubClassPackageName,
        subclass.id as SubClassId,
        subclass.name as SubClassName,
        inheritsfrom.depthofinheritance as DepthOfInheritance
FROM
        ttypes subclass,
        ttypes baseclass,
        tinheritances inheritsfrom,
        tpackages baseclasspackage,
        tpackages subclasspackage,
        TSourceEntities subsource,
        TFiles subfile,
        TSourceEntities basesource,
        TFiles basefile,
        TPackageContainmentRelations pcr
WHERE
        subclass.id = inheritsfrom.classid AND
        baseclass.id = inheritsfrom.superid AND
        baseclasspackage.id = baseclass.packageid AND
        subclasspackage.id = subclass.packageid AND

        /* Nur Source-Pakete analysieren */
        subsource.id = subclass.id AND
        subsource.sourcefileid = subfile.id AND
        basesource.id = baseclass.id AND
        basesource.sourcefileid = basefile.id AND

        /* Ist im Unterpaket vorhanden */
        baseclasspackage.id != subclasspackage.id AND
        baseclasspackage.fullName != '' AND
```

```
        subclasspackage.fullName != '' AND
        pcr.packageid =  baseclasspackage.id AND
        pcr.containingpackageid = subclasspackage.id AND

        /* Keine transitive Vererbung */
        inheritsfrom.depthofinheritance = 1

ORDER BY
        inheritsfrom.depthofinheritance
;
```

Lösungsstrategie

Es muss zuerst entschieden werden ob das Problem bei der Vererbungsbeziehung, bei der Paketstruktur oder aber die Platzierung der Klassen in den Paketen liegt. Wenn die Vererbungsbeziehung semantisch nicht begründet ist, muss diese eventuell durch Komposition ersetzt. Wenn die Paketaufteilung nicht die Spezialisierung der darin liegenden Klassen wiederpiegelt, muss eine neue Aufteilung gefunden werden und die Klassen entsprechen platziert werden. Wenn das Problem die falsche Platzierung der Klassen in den Paketen ist, müssen eine oder beide Klassen verschoben werden. Es ist zum Beispiel möglich, die Klasse des Oberpaketes in das Unterpaket, oder die Klasse des Unterpakets in das Oberpaket zu verschieben. Die Auswahl könnte durch Berücksichtigung von Kohäsionsmetriken gesteuert werden. Dabei muss auch auf Seiteneffekte geachtet werden, weil z.B. neue Pakethierarchie-Aufbrüche entstehen könnten.

2.1.33 Polymorphieplacebo

Definition

Eine statische Methode der Oberklasse `OK` ist in einer Unterklasse `UK` sichtbar, wird dort aber durch eine Methode mit der gleichen Signatur überdeckt.

Beschreibung

Ein wichtiger Verwendungszweck von Vererbung ist das Realisieren von Polymorphie zur Implementierung einer Typhierarchie: Die Oberklasse deklariert hierbei Schnittstellen und implementiert diese ggf. bereits. In den Unterklassen besteht die Möglichkeit, geerbte Methoden zu überdecken, um Spezifika der Unterklasse bei der Implementierung entsprechend zu berücksichtigen. Aufgrund der Polymorphie werden automatisch die Methoden derjenigen Unterklasse aufgerufen, dessen Instanz das Objekt darstellt.
Diese Möglichkeit, bestimmte Implementierungen in Abhängigkeit vom aktuellen Objekttyp auszuführen, besteht allerdings nur für Objektmethoden, nicht für Klassenmethoden (static). Wird eine statische Methode einer Oberklasse aufgerufen, so kann dieser Aufruf nicht durch eine Implementierung in der Unterklasse überdeckt werden. Es wird immer die Implementierung der Oberklasse ausgeführt, unabhängig davon, ob der Aufruf über Instanzen von Unterklassen erfolgt.
Obwohl die Vererbungsstruktur und die Überdeckungen eine Polymorphie suggerieren, werden die entsprechenden Methoden nicht polymorph aufgelöst (Polymorphieplacebo).

Programmiersprache

C++, Java, C#, ~~Delphi~~

Betrachtungsebene

Designebene

Informale Erkennung

- Suche nach statischen Methoden in Oberklassen, die in Unterklassen sichtbar sind.
- Suche nach Methoden in Unterklassen, die diese sichtbaren statischen Methoden überdecken.
- Jedes gefundene Methodenpaar (statische Oberklassenmethode und überdeckende Unterklassenmethode) stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: PolymorphiePlacebo.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:

Eine statische Methode der Oberklasse OK ist in einer Unterklasse UK sichtbar,
wird dort aber durch eine Methode mit der gleichen Signatur überdeckt.

Beschreibung:

Ein wichtiger Verwendungszweck von Vererbung ist das Realisieren von Polymorphie
(späte Bindung) zur Implementierung einer Typhierarchie: Die Oberklasse deklariert
hierbei Schnittstellen und implementiert diese ggf. bereits aus. In den
Unterklassen besteht die Möglichkeit, geerbte Methoden zu überdecken, um Spezifika
der Unterklasse bei der Implementierung entsprechend zu berücksichtigen (z.B. ein
spezieller Sortierungsalgorithmus). Ein Client dieser Typhierarchie benötigt
keinerlei Wissen um mögliche Unterklassen und deren spezifische Implementierung.
Er ist lediglich gegen die in der Superklasse deklarierten und evtl.
implementierten Methoden implementiert. Aufgrund der Vererbungsbeziehung ist der
Client damit automatisch in der Lage, auch mit Objekten von Unterklassen zu
arbeiten. Aufgrund der Polymorphie werden automatisch die Methoden derjenigen
Unterklasse aufgerufen, dessen Instanz das Objekt darstellt. Diese Möglichkeit,
bestimmte Implementierungen in Abhängigkeit vom aktuellen Objekttyp auszuführen,
besteht allerdings nur für Objektmethoden, nicht für Klassenmethoden (static).
Wird im Client gegen eine statische Methode der Superklasse programmiert, so wird
für diese Methode immer die Implementierung der Superklasse verwendet, unabhängig
davon, ob Objekte von Unterklassen verwendet werden und dort evtl. die Methoden
überdeckt sind. Obwohl die Vererbungsstruktur und die Überdeckungen eine
Polymorphie suggerieren, werden die entsprechenden Methoden nicht polymorph
aufgelöst (Polymorphieplacebo).

Implementierung:
- suche nach überschriebene geerbte statische methoden
*/


SELECT DISTINCT
        A.id AS SuperClassId,
        A.fullName AS SuperClassFullName,
        Am.id AS SuperClassMethodId,
        Am.name AS SuperClassMethodName,
        B.id AS SubClassId,
        B.fullName AS SubClassFullName,
        Bm.id AS PseudoOverrideMethodId,
        Bm.name AS PseudoOverrideMethodName
FROM
        TTypes AS A,
        TTypes AS B,
        TInheritances AS tInheritanceAB,
        TFunctions AS Am,
        TFunctions AS Bm,
        TSignatures AS SA,
        TSignatures AS SB,
        TMembers AS memberAm_A,
        TMembers AS memberBm_B,
        TConstants AS con
WHERE
        /* B erbt von A */
        tInheritanceAB.superId = A.id AND
        tInheritanceAB.classId = B.id AND
        Am.id = memberAm_A.id AND memberAm_A.classId = A.id AND
        Bm.id = memberBm_B.id AND memberBm_B.classId = B.id AND

        /* statische nicht private Methode */
        memberAm_A.isstatic = 1 AND
        memberAm_A.visibility != con.value AND
        con.name = 'VISIBILITY_PRIVATE' AND

        /* gleiche signatur */
        Am.id = SA.functionId AND
        Bm.id = SB.functionId AND
        SA.signature = SB.signature
;
```

Lösungsstrategie

Wenn die statische Natur der Methoden begründet und gewollt ist, empfiehlt es sich diesem unterschiedlichen Namen zu geben, um mögliche Missverständnisse in der Zukunft zu vermeiden. Im Gegenteil, wenn eine polymorphe Verwendung dieser Methode beabsichtigt wurde, muss das Statik-Attribut entfernt werden.

2.1.34 Potentielle Privatsphäre (Attribut)

Definition

Die deklarierte Sichtbarkeit eines Attributes (keine Konstanten; keine package-private-Sichtbarkeit) ist größer, als tatsächlich benötigt, d.h. das System ist auch nach einer Sichtbarkeitsreduktion noch übersetzbar.

Beschreibung

Das Sichtbarkeitskonzept in objektorientierten Sprachen erlaubt die Explizierung von Prioritäten beim Verstehen und Ändern von Systemen. Ein maximal sichtbares Programmierartefakt ist demzufolge besonders wichtig für das Verstehen der nach außen angebotenen Funktionalität; im Gegensatz dazu ist ein als privat deklariertes Artefakt für das primäre Verstehen des Zusammenspiels von Systemteilen weniger wichtig.
Die durch das Sichtbarkeitskonzept explizierte Priorisierung wird in diesem Qualitätsindikator für Attribute analysiert. Haben Attribute eine zu hohe Sichtbarkeit, d.h. aufgrund ihrer tatsächlichen Verwendung könnte ihre Sichtbarkeit auf "protected" oder "private" reduziert werden, so liegt eine falsche Priorisierung vor. Mit einer Einschränkung der Sichtbarkeit würden die Attribute eine geringere Priorität beim Verstehen und Ändern erhalten. Außerdem wären sie somit vor ungewollten Zugriffen geschützt, wodurch auch die Stabilität des Programms verbessert wird. Die Reduzierung der Sichtbarkeit hat weder einen Verlust von Informationen noch der Funktionalität zur Folge.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Eine Liste mit allen protected Attributen erstellen.
- Alle protected Attribute aus der Liste löschen, auf die durch andere Klassen zugegriffen wird (diese Attribute müssen protected sein/bleiben)
- Hinzufügen aller public Attribute zu der Liste
- Alle Attribute aus der Liste löschen, auf die von außerhalb der Vererbungshierarchie zugegriffen wird (diese müssen public bleiben)
- Die verbleibende Liste enthält alle Attribute, deren Sichtbarkeit eingeschränkt werden kann. Jedes Attribut stellt eine Probleminstanz dar.

Erkennung mit SISSy

```
/* $Id: PotentiellePrivatsphaereAttribut.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/* Potentielle Privatsphäre (Attribut) (ehem. Falsche Sichtbarkeit).

Definition:
 Die deklarierte Sichtbarkeit eines Attributes (keine Konstanten) ist größer,
 als tatsächlich benötigt, d.h. das System ist auch nach einer
 Sichtbarkeitsreduktion noch übersetzbar.

Beschreibung:
 Eine Klasse bietet Attribute öffentlich an, die entweder von der Klasse selbst
 oder nur von der eigenen Vererbungshierarchie dieser Klasse (aber nicht von
 aussen!) direkt gelesen oder/und geschrieben werden.
 oder
```

```
 Eine Klasse bietet Attribute protected an, die aber nur von der Klasse selbst
 zugegriffen werden

Implementation:
  (
    select all public attributes of A
      except
    select all public attributes of A that are accessed from classes that are not A and not
subclasses of A
  ) union (
    select all protected attributes of A
      except
    select all protected attributes of A that are accessed from classes other than A
  )
*/


(
/* select all public attributes of A */
SELECT DISTINCT
        A.id AS ClassContainingField_ID,
        A.fullName AS ClassContainingField_FullName,
        field.id AS Field_ID,
        field.name AS Field_FullName,
        memberAfield.visibility AS Field_Visibility
FROM
        TTypes AS A,
        TMembers AS memberAfield,
        TVariables AS field,
        TConstants AS constantField,
        TConstants AS constantPublic
WHERE
        /* joins */
        memberAfield.classId = A.Id AND
        memberAfield.id = field.Id AND

        /* public and non static-final attribute */
        field.KindOfVariable = constantField.Value AND
        constantField.Name = 'VAR_FIELD' AND
        NOT (memberAfield.isStatic = 1 AND memberAfield.isFinal = 1) AND
        memberAfield.visibility = constantPublic.Value AND
        constantPublic.Name = 'VISIBILITY_PUBLIC'

EXCEPT

/* select all public attributes of A that are accessed from classes that are not A and not
subclasses of A */

SELECT DISTINCT
        A.id AS ClassContainingField_ID,
        A.fullName AS ClassContainingField_FullName,
        field.id AS Field_ID,
        field.name AS Field_FullName,
        memberAfield.visibility AS Field_Visibility
FROM
        TTypes AS A,
        TMembers AS memberAfield,
        TVariables AS field,
        TTypes AS classM,
        TMembers AS memberClassM,
        TFunctions AS m,
        TConstants AS constantField,
        TConstants AS constantPublic,
        TAccesses AS access
WHERE
        /* joins */
        memberClassM.id = m.id AND
        memberClassM.classId = classM.id AND
        memberAfield.classId = A.Id AND
        memberAfield.id = field.Id AND
        access.targetId = field.id AND access.functionId = m.id AND

        /* public and non static-final attribute */
        field.KindOfVariable = constantField.Value AND
        constantField.Name = 'VAR_FIELD' AND
        NOT (memberAfield.isStatic = 1 AND memberAfield.isFinal = 1) AND
        memberAfield.visibility = constantPublic.Value AND
        constantPublic.Name = 'VISIBILITY_PUBLIC' AND

        /* ...are accessed from classes that are not A and not subclasses of A */
```

```
        A.id <> classM.id AND
        NOT EXISTS (
                SELECT *
                FROM TInheritances i
                WHERE i.superId = A.id AND i.classId = classM.id
        )

) UNION (

/* select all protected attributes of A */
SELECT DISTINCT
        A.id AS ClassContainingField_ID,
        A.fullName AS ClassContainingField_FullName,
        field.id AS Field_ID,
        field.name AS Field_FullName,
        memberAfield.visibility AS Field_Visibility
FROM
        TTypes AS A,
        TMembers AS memberAfield,
        TVariables AS field,
        TConstants AS constantField,
        TConstants AS constantProtected
WHERE
        /* joins */
        memberAfield.classId = A.Id AND
        memberAfield.id = field.Id AND

        /* protected and non static-final attribute */
        field.KindOfVariable = constantField.Value AND
        constantField.Name = 'VAR_FIELD' AND
        NOT (memberAfield.isStatic = 1 AND memberAfield.isFinal = 1) AND
        memberAfield.visibility = constantProtected.Value AND
        constantProtected.Name = 'VISIBILITY_PROTECTED'

EXCEPT

/* select all protected attributes of A that are accessed from classes other than A */
SELECT DISTINCT
        A.id AS ClassContainingField_ID,
        A.fullName AS ClassContainingField_FullName,
        field.id AS Field_ID,
        field.name AS Field_FullName,
        memberAfield.visibility AS Field_Visibility
FROM
        TTypes AS A,
        TMembers AS memberAfield,
        TVariables AS field,
        TTypes AS classM,
        TMembers AS memberClassM,
        TFunctions AS m,
        TConstants AS constantField,
        TConstants AS constantProtected,
        TAccesses AS access
WHERE
        /* joins */
        memberClassM.id = m.id AND
        memberClassM.classId = classM.id AND
        memberAfield.classId = A.Id AND
        memberAfield.id = field.Id AND
        access.targetId = field.id AND access.functionId = m.id AND

        /* protected and non static-final attribute */
        field.KindOfVariable = constantField.Value AND
        constantField.Name = 'VAR_FIELD' AND
        NOT (memberAfield.isStatic = 1 AND memberAfield.isFinal = 1) AND
        memberAfield.visibility = constantProtected.Value AND
        constantProtected.Name = 'VISIBILITY_PROTECTED' AND

        /* ...are accessed from classes other than A */
        A.id <> classM.id
);
```

Lösungsstrategie

Die Sichtbarkeit des Attributs muss neu überdacht werden und bei Bedarf entsprechend der Verwendung des Attributs abgeändert werden. Eine zu restriktive Reduzierung führt zu einem Compilefehler, die sehr schnell ermittelt werden kann und für den keine Testläufe notwendig sind.

2.1.35 Potentielle Privatsphäre (Methode)

Definition

Eine protected Methode einer Klasse K wird in keiner Unterklasse von K benutzt oder überschrieben und überschreibt selbst keine Methode einer Oberklasse von K.

Beschreibung

Methoden implementieren das Verhalten einer Klasse und bilden gleichzeitig deren Schnittstelle. Je größer (umfangreicher) eine Schnittstelle ist, umso schwieriger/aufwendiger wird es, die Klasse zu verstehen und zu modifizieren. Daher sollte diese Schnittstelle so klein wie möglich gehalten werden.
Dieses Problemmuster fokussiert diejenigen Methoden der Schnittstelle, die für Unterklassen zur Verfügung gestellt werden sollen. Auch dieser Teil ist minimal zu halten. Diese Forderung ist für eine Schnittstelle verletzt, wenn eine Schnittstellenmethode in keiner Unterklasse überschrieben oder aufgerufen wird und wenn die Schnittstellenmethode selbst keine Methode einer Oberklasse überschreibt oder implementiert.
In Java gibt es den Spezialfall, dass auch von außerhalb der Vererbungshierarchie auf protected Methoden zugegriffen werden kann. Dies ist dann möglich, wenn sich die aufrufende Klasse im gleichen Paket befindet. Diese Aufrufe könnten zwar für Unit-Tests verwendet werden[1], diese Zugriffsmöglichkeit widerspricht jedoch dem klassischen Verständnis von protected Methoden. Zugriffe auf protected Methoden werden daher nicht erlaubt (siehe Qualitätsindikator „Insiderwissen“). Die angesprochenen JUnit-Tests können stattdessen über die Verwendung der "default" Sichtbarkeit ermöglicht werden.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Suche aller protected Methoden, die in Unterklassen nicht benutzt und auch nicht überschrieben werden.
- Die Betrachtung gilt nicht bei protected Methoden die selber eine protected Methode überschreiben oder implementieren.
- Jede protected Methode, deren Sichtbarkeit eingeschränkt werden kann, wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: PotentiellePrivatsphaereMethode.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */

CREATE OR REPLACE VIEW VppsMethod AS

(
SELECT
        function.id,
        function.name,
        function.classid
FROM
        TFunctions function join
        TConstants cons on (function.kindoffunction = cons.value) join
        TMembers member on (member.id = function.id) join
        TConstants cons2 on (member.visibility = cons2.value)
WHERE
        cons.name = 'FUNC_METHOD' AND
        cons2.name ='VISIBILITY_PROTECTED'

/*
Methoden einer Oberklasse, die von der Unterklasse aufgerufen werden
```

[1] Eine Empfehlung bei der Verwendung von JUnit besagt, dass Testklassen zum gleichen Paket wie die zu testende Klasse gehören sollten.

```
*/

EXCEPT

SELECT
        function.id,
        function.name,
        function.classid
FROM
        TFunctions function join
        TTypes type on (function.classid = type.id) join
        TInheritances inheritance on (inheritance.superid = type.id) join
        TTypes subType on (inheritance.classid = subtype.id) join
        TAccesses access on (function.id = access.targetid and access.classid = subtype.id)


EXCEPT

/* Methoden, die in einer Unterklasse überschrieben werden */
SELECT
        function.id,
        function.name,
        function.classid
FROM
        TFunctions function join
        TTypes type on (function.classid = type.id) join
        TInheritances inheritance on (inheritance.superid = type.id) join
        TTypes subType on (inheritance.classid = subtype.id) join
        TSignatures         signaturOverriddenMethod         on         (function.id         =
signaturOverriddenMethod.functionid) join
        TSignatures   signatureOverridingMethod   on   (signaturOverriddenMethod.signature   =
signatureOverridingMethod.signature) join
        TFunctions          overridingmethod          on          (overridingMethod.id         =
signatureOverridingMethod.functionid)
WHERE
        overridingmethod.classid = subType.id


EXCEPT

/* Methoden, die eine Methode einer Oberklasse überschreiben */
SELECT
        function.id,
        function.name,
        function.classid
FROM
        TFunctions function join
        TTypes type on (function.classid = type.id) join
        TInheritances inheritance on (inheritance.classid = type.id) join
        TTypes superType on (inheritance.superid = supertype.id) join
        TSignatures         signaturOverridingMethod         on         (function.id         =
signaturOverridingMethod.functionid) join
        TSignatures   signatureOverridingMethod   on   (signaturOverridingMethod.signature   =
signatureOverriddenMethod.signature) join
        TFunctions          overriddenmethod          on          (overriddenMethod.id         =
signatureOverriddenMethod.functionid)
WHERE
        overriddenmethod.classid = superType.id

)
;

SELECT
                function.id,
                function.name,
                function.classid,
                type.fullname,
                file.pathname
FROM
        VppsMethod function join
        TSourceEntities se on (function.id = se.id) join
        Tfiles file on (file.id = se.sourcefileid) join
        TTypes type on (type.id = function.classid)
;

drop view VppsMethod;
```

Lösungsstrategie

Die Sichtbarkeit von in Unterklassen nicht referenzierten protected Methoden muss überdacht, und bei Bedarf eingeschränkt werden. Im Fall von Java ist zu prüfen, ob die Methode nicht doch für Unit-Tests verwendet wird. In diesem Fall sollte die Sichtbarkeit auf "default" geändert werden. Anderenfalls wird die Sichtbarkeit der Methode auf privat gesetzt.

2.1.36 Pränatale Kommunikation

Definition

Im Konstruktor einer Klasse K wird mindestens eine virtuelle Methode von K (auch geerbte) direkt aufgerufen, unabhängig davon, ob die virtuelle Methode in Unterklassen bereits überschrieben wird oder in K bereits implementiert ist.

Beschreibung

Virtuelle Methoden werden verwendet, um Polymorphie zu ermöglichen, d.h. das Versenden einer Nachricht führt zum Ausführen einer Methode derjenigen Klasse, dessen Typ das Objekt hat, das die Nachricht erhält. Polymorphe Aufrufe sind auch innerhalb von Konstruktoren möglich, dort aber häufig sehr gefährlich, da die Objekte, die die Nachricht empfangen, noch nicht vollständig instanziiert sind.
Grundsätzlich gefährlich wird pränatale Kommunikation besonders dann, wenn in der virtuellen Methode der Unterklasse UK Spezifika der Unterklasse UK verwendet werden. Ist dies nicht der Fall, muß die pränatale Kommunikation nicht zu Fehlern führen. Da solche Besonderheiten allerdings während der Wartung nicht langfristig gewährleistet werden können, sind solche Konstrukte zu vermeiden.

Programmiersprache

C++, Java, ~~C#~~, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Suche nach Konstruktoren, die eine in derselben Klasse verfügbare virtuelle Methode aufrufen.
- Jeder Konstruktor, der mindestens eine virtuelle Methode aufruft, wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: PraenataleKommunikation.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:

Im Konstruktor einer Klasse K werden virtuelle Methoden aus K (auch geerbte)
direkt oder indirekt aufgerufen, unabhängig davon, ob die virtuelle Methode in
Unterklassen bereits überschrieben wird.

Beschreibung:

Virtuelle Methoden werden verwendet, um Polymorphie zu ermöglichen, d.h. das
Versenden einer Nachricht führt zum Ausführen einer Methode derjenigen Klasse,
dessen Typ das Objekt, das die Nachricht erhält, hat. Polymorphe Aufrufe sind
auch innerhalb von Konstruktoren möglich, dort aber häufig sehr gefährlich, da
die Objekte, die die Nachricht empfangen, noch nicht vollständig instanziiert
sind. Ein Beispiel ist die Superklasse Mensch mit dem Attribut Geburtsjahr und
der Unterklasse Student mit dem Attribut Immatrikulationsnummer. Besitzen beide
Klasse eine virtuelle Init()-Methode, innerhalb derer die jeweiligen Attribute
auf einen initialen Wert gesetzt werden, und wird diese Init()-Methode innerhalb
des jeweiligen Konstruktors aufgerufen, so besteht die Gefahr der pränatalen
```

```
Kommunikation: Beim Erzeugen eines neuen Studenten-Objekts wird zuerst der
Konstruktor der Klasse Mensch aufgerufen. Der dortige Init()-Aufruf wird
allerdings bereits polymorph auf den Studenten abgebildet. Das dortige
Bearbeiten des Attributes Immatrikulationsnummer führt allerdings zum Absturz,
da das Objekt Student noch nicht vollständig instanziiert ist ("pränatal"), d.h.
insbesondere deren Attribute noch nicht korrekt im Speicher abgelegt sind.
Prinzipiell tritt pränatale Kommunikation besonders dann auf, wenn in der
virtuellen Methode der Unterklasse UK Spezifika der Unterklasse UK verwendet
werden (vgl. Beispiel oben). Ist dies nicht der Fall, muß die pränatale
Kommunikation nicht zu Fehlern führen. Da solche Besonderheiten allerdings
während der Wartung nicht langfristig gewährleistet werden können, sind auch
solche Konstrukte zu vermeiden.

Implementierung:


*/

SELECT DISTINCT
       A.id AS SuperClassId,
       A.fullName AS SuperClassFullName,
       Am_Con.id AS SuperClassConstructorId,
       Am_Con.name AS SuperClassConstructorName,
       sig.signature as signature
FROM
       TTypes A  inner join
       TFunctions AS Am_Con on (Am_Con.classid = A.id) inner join
       Taccesses as accesses on (accesses.functionId = Am_con.id) inner join
       TFunctions AS Am on (accesses.targetid = Am.id and Am.classId = A.id) inner join
       TMembers AS funcMember on (funcMember.id = Am.id AND funcMember.isfinal = 0) inner
join
       TSignatures sig on (sig.functionid = Am_con.id),
       TConstants con,
       TConstants con2
WHERE
       Am_con.kindoffunction = con.value  AND
       con.name = 'FUNC_CONSTRUCTOR' AND

       /* Konstruktor der Oberklasse muss diese Methode aufrufen */

       con2.value = Am.kindoffunction AND
       con2.name = 'FUNC_METHOD'



;
```

Lösungsstrategie

Die Behebung des Problems besteht im Allgemeinen aus den folgenden Möglichkeiten:

- Ermittlung der Unterschiede der Methoden, die die pränatale Kommunikation bewirken. Häufig sind die Methoden inhaltlich sehr ähnlich und können in die Oberklasse zusammengeführt werden.
- Herausfaktorisieren der virtuellen Aufrufe aus dem Konstruktor. Damit obliegt es dem Client, die Methoden aufzurufen. Da er aber nur mit fertigen Objekten sprechen kann, existiert keine Gefahr der pränatalen Kommunikation.

2.1.37 Risikocode

Definition

Typische Techniken der defensiven Programmierung (Fail-safe-Mechanismen) werden falsch eingesetzt. Hierzu zählen für den Code-Sünden-Index leere exception-Handler sowie fehlende break-und default-Statements bei Switch-Statements.

Beschreibung

Fail-safe Mechanismen wie Exception-Handling und default-Pfade für switch-Anweisungen sorgen dafür, dass die Anwendung bei unvorhergesehenen Ereignissen (z.B. andere Eingabewerte) den Entwickler direkt von denjenigen Stellen in Kenntnis setzt, bei denen die Anomalie aufgetreten ist. Werden diese Mechanismen

vergessen (z.B. fehlendes default) oder vorsätzlich ausgehebelt (z.B. leerer Exception-Handler), führt dies dazu, dass evtl. notwendige Änderungen des Programms verschleiert werden und das System u.U. viel später an ganz anderen Stellen falsche Werte liefert.
Ein leerer Catch - Block führt z.B. dazu, dass alle Ausnahmen eines bestimmten Typs ignoriert werden, unabhängig davon, was die Ausnahmen tatsächlich ausgelöst hat. Problematisch ist, dass nur Annahmen über etwas gemacht werden kann, was auch vorausgesehen werden kann. Ein leerer Catch-Block unterscheidet aber nicht zwischen vorhergesehenen und unvorhergesehenen Ausnahmen, sondern ignoriert alle. Fehlende Default-Pfade können dazu führen, dass das Programm aufgrund geänderter Eingabedaten in einen undefinierten Zustand gerät, der erst viel später im Programmfluss offenkundig wird. Bei sicheren Abfragen aller möglichen Werte ist z.B. ein default: ASSERT(FALSE) hilfreich.
Fehlende Breaks auf der anderen Seite bewirken einen besonderen Kontrollfluss, der nicht von demjenigen zu unterscheiden ist, bei dem das break vergessen wurde.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

- Zählen des Deltas zwischen switch-Anweisungen und switch-default Paaren und
- zählen des Deltas zwischen case-Anweisungen und case-break Paaren und
- zählen leerer Exception-Handler. Ein leerer Exception-Handler liegt vor, wenn dieser kein einziges Statement bzw. Kommentarzeile enthält.
- Jedes fehlende default und break bzw jede leere Exception Anweisung wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: RisikoCode.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:

Typische Techniken der defensiven Programmierung (Fail-safe-Mechanismen) werden
falsch eingesetzt. Hierzu zählen leere exception-Handler sowie fehlende break-
und default-Statements bei Switch-Statements.

Beschreibung:

Fail-safe Mechanismen wie Exception-Handling und break-Statements oder default-
Pfade für switch-Anweisungen sorgen dafür, dass die Anwendung bei
unvorhergesehenen Ereignissen (z.B. andere Eingabewerte) den Entwickler direkt
von denjenigen Stellen in Kenntnis setzt, bei denen die Anomalie aufgetreten
ist. Werden diese Mechanismen vergessen (z.B. fehlendes default) oder
vorsätzlich ausgehebelt (z.B. leerer Exception-Handler), führt dies dazu, dass
evtl. notwendige Änderungen des Programms verschleiert werden und das System
u.U. viel später an ganz anderen Stellen falsche Werte liefert. Ein leerer Catch
- Block führt z.B. dazu, dass alle Ausnahmen eines bestimmten Typs ignoriert
werden, unabhängig davon, was die Ausnahmen tatsächlich ausgelöst hat.
Problematisch ist, dass wir nur Annahmen über etwas machen können, dass wir auch
voraussehen können. Ein leerer Catch-Block unterscheidet aber nicht zwischen
vorhergesehenen und unvorhergesehenen Ausnahmen, sondern ignoriert alle.
Fehlende Default-Pfade können dazu führen, dass das Programm aufgrund geänderter
Eingabedaten in einen undefinierten Zustand gerät, der erst viel später im
Programmfluss offenkundig wird. Bei sicheren Abfragen aller möglichen Werte ist
z.B. ein default: ASSERT(FALSE) hilfreich. Fehlende Breaks bewirken einen
besonderen Kontrollfluss, der nicht von demjenigen zu unterscheiden ist, bei dem
das break vergessen wurde.

Implementierung:

- Suche nach leeren Catch Blocks (die keine Statements beinhalten)
*/

CREATE OR REPLACE VIEW VNumberOfSourceStatementsPerBranch AS
```

```
SELECT
        branchStatement.id,
        COUNT(branchStatement.Id)
FROM
        TStatements branchStatement join
        TStatements childStatement on (childStatement.parentStatementid = branchStatement.id)
join
        TConstants con on (con.value = branchStatement.kindofStatement) join
        TSourceEntities se on (se.id = childStatement.id)
WHERE
        con.name = 'STATEMENT_BRANCH'
GROUP BY
        branchStatement.id
;



SELECT DISTINCT
        ecb.id AS IDEmptyCatchBlock,
        ecb.functionId AS IDContainingFunction,
        f.name AS NameContainingFunction,
        f.classId AS IDContainingClass,
        C.name AS NameContainingClass,
        'EMPTYCATCHBLOCK',
        files.pathname
FROM
        TStatements AS ecb,
        TFunctions AS f,
        TTypes AS C,
        TConstants AS constCatchBlock,
        TSourceEntities sourceEntities,
        TFiles files

WHERE
        /* look for catch blocks */
        ecb.kindOfStatement = constCatchBlock.value AND
        constCatchBlock.name = 'STATEMENT_CATCHBLOCK' AND

        /* some joins */
        ecb.functionId = f.id AND
        f.classId = C.id AND

        C.id = sourceEntities.id AND
        sourceEntities.sourcefileid = files.id


/* take out non-empty catch blocks */
EXCEPT

SELECT DISTINCT
        ecb.id AS IDEmptyCatchBlock,
        ecb.functionId AS IDContainingFunction,
        f.name AS NameContainingFunction,
        f.classId AS IDContainingClass,
        C.name AS NameContainingClass,
        'EMPTYCATCHBLOCK',
        files.pathname
FROM
        TStatements AS ecb,
        TStatements AS ecbMemberStatement,
        TFunctions AS f,
        TTypes AS C,
        TConstants AS constCatchBlock,
        TSourceEntities sourceEntities,
        TFiles files
WHERE
        /* look for catch blocks */
        ecb.kindOfStatement = constCatchBlock.value AND
        constCatchBlock.name = 'STATEMENT_CATCHBLOCK' AND

        /* some joins */
        ecb.functionId = f.id AND
        f.classId = C.id AND
        ecbMemberStatement.parentStatementId = ecb.id AND

        C.id = sourceEntities.id AND
        sourceEntities.sourcefileid = files.id

UNION
```

```
SELECT
        branchStatement.id,
        func.id,
        func.name,
        type.id,
        type.name,
        'MISSINGDEFAULT',
        files.pathname
FROM
        TStatements branchStatement join
        TStatements branch on (branch.parentstatementid = branchstatement.id) join
        TConstants con on (branchStatement.kindofstatement = con.value) join
        TFunctions func on (branchStatement.functionid = func.id) join
        TTypes type on (func.classid = type.id) join
        VNumberOfSourceStatementsPerBranch sspb on (sspb.id = branchStatement.id) join
        TSourceEntities sourceEntities on (type.id = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id)
WHERE
        con.name = 'STATEMENT_BRANCH' AND
        sspb.count > 1 AND

        NOT EXISTS (
                SELECT
                        *
                FROM
                        TSourceEntities sEntity
                Where
                        sEntity.id = branch.id
        )

;

DROP VIEW VNumberOfSourceStatementsPerBranch;
```

Lösungsstrategie

Fehlende break-Statements und default-Pfaden müssen ergänzt werden.
Bei leeren Catch-Blöcke, muss man überprüfen wo die Zuständigkeit für Ausnahmesituationen für den Code im try-block liegt:

- Wenn diese Zuständigkeit nicht in der Methode wo der try-Block steht, sondern höher liegt, empfiehlt es sich den try-catch-Block zu entfernen und alle Ausnahmesituationen „weiter oben“ zu behandeln.
- Wenn die Zuständigkeit nur teilweise lokal liegt, ist es empfehlenswert den try-catch Block durch eine if Anweisung zu ersetzten, die die Vorbedingungen des kritischen Codes vor der Ausführung überprüft. So können die Ausnahmesituationen, für die eine Zuständigkeit existiert, behandelt werden. Weitere Ausnahmefälle werden wie in der ersten Lösungsalternative nach oben weitergeleitet. Bei mehreren gleichzeitigen threads muss man aufpassen, denn es besteht theoretisch die Möglichkeit dass zwischen der Überprüfung der Vorbedingungen und der Ausführung des kritischen Codes, eine Änderung im Zustand vorkommt.
- Wenn die Zuständigkeit für die Behandlung von Ausnahmesituationen lokal liegt, muss ein entsprechender Catch-Block implementiert werden. Auch wenn es sicher ist, dass keine Ausnahmesituationen vorkommen sollten ist es empfehlenswert eine ASSERT(false) oder zumindest eine Logging-Anweisung im Catch-Block zu platzieren.

2.1.38 Signaturähnliche Klassen

Definition

Mindestens zwei öffentliche, nicht-geschachtelte Klassen, die weder in einer (in)direkten Vererbungsbeziehung stehen, noch eine gemeinsame (in)direkte Oberklasse haben, besitzen mehr als 50% oder mind. 10 identische, nicht-geerbte und nicht in der Oberklasse/Interface deklarierte, öffentliche Methodensignaturen. Struktoren werden weder bei der Methodenfindung noch bei der Ermittlung des Prozentwertes berücksichtigt. Existiert eine direkte oder indirekte Oberklasse/Interface in der Library, so dürfen die Methodensignaturen dort nicht bekannt sein.

Beschreibung

Vererbung ist ein wichtiges Konzept, um Ähnlichkeiten zwischen verschiedenen Klassen zu explizieren. Ist dies geschehen, so entstehen dadurch höherwertige Klassencluster, die jeweils zusammenhängend verstanden, gewartet und getestet werden können. Ein wichtiger Aspekt der Ähnlichkeit sind identische Signaturen bei Methoden.
Existieren Klassen, die bzgl. Vererbung in keinerlei direkter oder indirekter Relation stehen, aber dennoch viel Ähnlichkeiten aufweisen, so sind die Vorteile der Vererbung für diese Klasse nicht genutzt. Vielmehr resultieren daraus Risiken der Mehrfachwartung, des Mehrfachtestens und der Duplikation mit anschließender Variantenbildung. Insbesondere das Fehlen von explizit gemachten Klassenclustern wirkt sich deutlich negativ auf das Verstehen und Verändern großer Systeme aus.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Architekturebene

Informale Erkennung

- Suche nach Paaren von Klassen, die die gleichen Signaturen enthalten.
- Um falsch positive bestmöglich zu verhindern, müssen mehr als 50% oder mindestens 10 der Methoden signaturgleich sein.
- Jedes gefundene Klassenpaar wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* Bestimmen der Anzahl der öffentlichen Methoden pro Klasse*/
create table TPublicMethodsPerClass as

select
        type.id,
        count (function.id) as count
from
        TTypes type join
        TFunctions function on (type.id = function.classid) join
        TMembers funcMember on (function.id = funcMember.id) join
        TConstants con on (funcMember.visibility = con.value) join
        TConstants con2 on (function.kindoffunction = con2.value)
where
        con.name = 'VISIBILITY_PUBLIC' and
        con2.name = 'FUNC_METHOD'

group by
        type.id
;

/* Bestimme für jedes Klassenpaar, die Anzahl der öffentlichen Methoden, deren Signaturen
übereinstimmen*/
create Table TCommonMethodsPerClassPair as

select
        clsA.id as Class_A_Id,
        clsB.id as Class_B_id,
        count (funcA.id)
from
        TTypes clsA
inner join
        TTypes clsB on (clsB.id > clsA.id and clsB.classId != clsA.id and clsA.classId !=
clsB.id)
inner join
        TMembers funcA on clsA.id = funcA.classId
inner join
        TSignatures sigA on funcA.id = sigA.functionId
inner join
        TSignatures sigB on (sigA.signature = sigB.signature and sigA.functionId !=
sigB.functionId)
inner join
        TMembers funcB on (clsB.id = funcB.classId and sigB.functionId = funcB.id)
```

```
inner join
       TConstants con on (con.value = funcA.kindOfMember and con.name = 'FUNC_METHOD') join
       TConstants con2 on (con2.value = funcA.visibility and con2.name = 'VISIBILITY_PUBLIC')
join
       TConstants con3 on (con3.value = funcB.visibility and con3.name = 'VISIBILITY_PUBLIC')

where
       clsA.classid = -1 and
       clsB.classid = -1
group by
       clsA.id,
       clsB.id
;

/* Bestimme die potentiellen Trefferpaare, also alle Paare von Klassen, die nicht in einer
(in)direkten Vererbungsbeziehung zueinander stehen, d.h. die auch keine gemeinsame Oberklasse
haben */
create table TPotentialPairs as

select
       clsA.id as Class_A_Id,
       clsB.id as Class_B_Id,
       clsA.fullname as aname,
       clsB.fullname as bname
from
       TTypes clsA
inner join
       TTypes clsB on (clsB.id > clsA.id and clsB.classId != clsA.id and clsA.classId !=
clsB.id)
where
       clsA.classid = -1 and
       clsB.classid = -1


except
(
select
       clsA.id as Class_A_Id,
       clsB.id as Class_B_Id,
       clsA.fullname as aname,
       clsB.fullname as bname

from
       TTypes clsA
inner join
       TTypes clsB on (clsB.id > clsA.id and clsB.classId != clsA.id and clsA.classId !=
clsB.id)
inner join
       TInheritances inh on ((clsB.id = inh.classId and inh.superId = clsA.id) or (clsB.id =
inh.superId and inh.classId = clsA.id))
where
       clsA.classid = -1 and
       clsB.classid = -1
UNION

select
       clsA.id as Class_A_Id,
       clsB.id as Class_B_Id,
       clsA.fullname as aname,
       clsB.fullname as bname
from
       TTypes clsA
inner join
       TTypes clsB on (clsB.id > clsA.id and clsB.classId != clsA.id and clsA.classId !=
clsB.id)
inner join
       TInheritances inh2 on  (inh2.classid=clsA.id)
inner join
       TInheritances inh3 on  (inh3.classid=clsB.id)
inner join
       TTypes  superType on (superType.id = inh2.superid and superType.id = inh3.superid)
where
       clsA.classid = -1 and
       clsB.classid = -1
       and inh2.superId=inh3.superId
       AND superType.fullname <> 'java.lang.Object'
)
;
```

```
/* Bestimme für die potentiellen Paare die Anzahl der öffentlichen Methoden mit identischen
Signaturen */
create table TNumberOfIdenticalSignatures as

select
        sa.id as Class_A_id,
        sb.id as Class_B_id,
        pp.aname,
        pp.bname,
        cmpc.count as numberMethodSignatures
from
        TPotentialPairs pp join
        TCommonMethodsPerClassPair cmpc on (pp.Class_A_Id = cmpc.Class_A_Id and pp.Class_B_id
= cmpc.Class_B_id) join
        TSourceEntities as sA on (sA.id = cmpc.Class_A_Id) join
        TSourceEntities as sB on (sB.id = cmpc.Class_B_Id)
where
        sA.sourceFileId >= 0 and
        sB.sourceFileId >= 0
;


/* Alle Paare ausgeben, die mehr als 50% identische Methoden haben */
select
        *
from
        TNumberOfIdenticalSignatures nois join
        TPublicMethodsPerClass pmpcA on (nois.Class_A_id = pmpcA.id) join
        TPublicMethodsPerClass pmpcB on (nois.Class_B_id = pmpcB.id)
where
   nois.NumberMethodSignatures >= 10 or
   (
      ((CAST(nois.NumberMethodSignatures as real) / pmpcA.count * 100) > 50) and
      ((CAST(nois.NumberMethodSignatures as real) / pmpcB.count * 100) > 50)
   )
;

drop table TPublicMethodsPerClass;
drop table TCommonMethodsPerClassPair;
drop table TPotentialPairs;
drop table TNumberOfIdenticalSignatures;
```

Lösungsstrategie

Die Behebung des Problemmusters signaturähnliche Klassen geschieht entlang der folgenden Schritte:

- Schaffung einer Schnittstelle oder bei Bedarf Oberklasse, die die gemeinsamen Signaturen enthält.
- Integration der signaturähnlichen Klassen in eine Vererbungsstruktur mit der neuen Schnittstelle / Oberklasse.
- Exploration, inwieweit die Signaturen der Oberklasse mit einer Implementierung, die allen Unterklassen gemeinsam ist, angereichert werden kann.
- Anpassung der Clients der signaturähnlichen Klassen gegen die neue Schnittstelle.

2.1.39 Simulierte Polymorphie

Definition

Innerhalb einer Methode wird ein Objekt auf mehrere Objekttypen überprüft. Für Problemmusteridentifikation wird im Fall von Java auf die Suche nach `instanceof` und in C++ auf die Suche nach `typeid` beschränkt.

Beschreibung

Die Typsicherheit der hier betrachteten objektorientierten Sprachen stellt sicher, dass an einen Bezeichner im Quelltext nur diejenigen Nachrichten geschickt werden können, die der Typ der Deklaration besitzt. Durch die Deklaration eines Bezeichners mit dem Typ eines Interfaces kann hierdurch z.B. sichergestellt werden, dass nur die Funktionalität des Interfaces verwendet wird. Die in all diesen Fällen erreichte bewusste Unabhängigkeit

vom tatsächlichen Typ zur Laufzeit kann durch Typprüfungen umgangen werden (in Java z.B. mittels der instanceof-Funktionalität).
Wird ein Objekt zur Laufzeit nach seinem Typ gefragt und werden in Abhängigkeit davon unterschiedliche Folge-Operationen aufgerufen, so kann hierdurch Polymorphie simuliert werden. Im Gegensatz zur echten Polymorphie, bei der das Laufzeitsystem den tatsächlichen Adressat einer Nachricht ermittelt und der Programmierer die Unterscheidungen nicht mehr statisch im Code ablegen muss, ist diese simulierte Polymorphie allerdings sehr wartungsunfreundlich. So muss ein neuer Adressat einer Nachricht an allen Stellen mit simulierter Polymorphie bekannt gemacht werden. Auch jegliche Änderung der Adressaten (z.B. Name und Ort) müssen mehrfach dezentral durchgeführt werden. Die durch die simulierte Polymorphie erzeugten Strukturen stellen insgesamt ein großes Risiko für die langfristige Zukunftsfähigkeit eines Systems dar.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

- Finde alle Methoden, in denen ein und dasselbe Objekt auf mehrere Objekttypen hin überprüft wird. In JAVA wird erfolgt die Überprüfung mittels instanceof und in C++ mittels typeid.
- Jede so gefundene Methode wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: SimuliertePolymorphie.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:

Innerhalb einer Methode wird ein Objekt auf mehrere Objekttypen überprüft.

Beschreibung:

Die Typsicherheit der hier betrachteten objektorientierten Sprachen stellt
sicher, dass an einen Bezeichner im Quelltext nur diejenigen Nachrichten
geschickt werden können, die der Typ der Deklaration besitzt. Durch die
Deklaration eines Bezeichners mit dem Typ eines Interfaces kann hierdurch z.B.
sichergestellt werden, dass nur die Funktionalität des Interfaces verwendet
wird. Die in all diesen Fällen erreichte bewusste Unabhängigkeit vom
tatsächlichen Typ zur Laufzeit kann durch Typprüfungen umgangen werden (in Java
z.B. mittels der instanceof-Funktionalität). Wird ein Objekt zur Laufzeit nach
seinem Typ gefragt und in Abhängigkeit davon unterschiedliche Folge-Operationen
aufgerufen, so kann hierdurch Polymorphie simuliert werden. Im Gegensatz zur
echten Polymorphie, bei der das Laufzeitsystem den tatsächlichen Adressat einer
Nachricht ermittelt und der Programmierer die Unterscheidungen nicht mehr
statisch im Code ablegen muss, ist diese simulierte Polymorphie allerdings sehr
wartungsunfreundlich. So muss ein neuer Adressat einer Nachricht an allen
Stellen mit simulierter Polymorphie bekannt gemacht werden. Auch jegliche
Änderung der Adressaten (z.B. Name und Ort) müssen mehrfach dezentral
durchgeführt werden. Die durch die simulierte Polymorphie erzeugten Strukturen
stellen insgesamt ein großes Risiko für die langfristige Zukunftsfähigkeit eines
Systems dar.

Implementierung:

look for methods in which:
- min 2 RTTI type accesses in branch statements
- referred type is ancestor for the checked type and both are defined in the
system itself

BUG: displayed number of type checks sometimes too high, due to read accesses that are also
included in the count. Replaced by an intensity index

*/


create or replace view VRTTIAccesses as
```

```
select distinct
       C.id as ClassId,
       C.fullName as ClassFullName,
       f.id as FunctionId,
       f.name as FunctionName,
       statement.id as StatementId,
       Sub.id as ReqTypeId,
       acSub.id as AccessId,
       acSub.position as position
from
       TConstants as constRTTIAc inner join
       TAccesses as acSub on (constRTTIAc.name = 'TYPEACCESS_RTTI' and acSub.kindOfAccess =
constRTTIAc.value) inner join
       TFunctions as f on (f.id = acSub.functionId) inner join
       TTypes as Sub on Sub.id = acSub.targetId inner join
       TTypes as C on (C.id = acSub.classId) inner join
       TStatements as statement on acSub.statementID = statement.id,
       TConstants as constBranchSt
;

SELECT
       RTTIAc.ClassId as IDContainingClass,
       RTTIAc.ClassFullName as NameContainingClass,
       RTTIAc.FunctionId as IDContainingMethod,
       RTTIAc.FunctionName as NameContainingMethod,
       v.name as NameVariable,
       COUNT(*) as IntensityIndex
FROM
       VRTTIAccesses as RTTIAc inner join
       TAccesses as acSuper on (RTTIAc.ReqTypeId = acSuper.targetid) inner join
       TConstants as constDeclAc on (constDeclAc.value = acSuper.kindofaccess) inner join
       TConstants as constVarReadAc on      constVarReadAc.name = 'VARACCESS_READ' inner join
       TAccesses as acV on (RTTIAc.statementid = acV.statementid AND RTTIAc.position - 1 =
acV.position AND acV.kindOfAccess = constVarReadAc.value) inner join
       TVariables as v on (acV.targetId = v.id ) inner join
       TSourceEntities as seSub on (RTTIAc.ReqTypeId = seSub.id and seSub.sourceFileID > 0)
inner join
       TSourceEntities as seSuper on (RTTIAc.ReqTypeId = seSuper.id and seSuper.sourceFileID
> 0)
GROUP BY
       RTTIAc.ClassId,
       RTTIAc.ClassFullName,
       RTTIAc.FunctionId,
       RTTIAc.FunctionName,
       v.name

HAVING COUNT(*) > 1

ORDER BY 6 DESC

;


drop view VRTTIAccesses;
```

Lösungsstrategie

Die Behebung dieses Problemmusters besteht in der Anwendung der Polymorphie, entlang der folgenden Schritte:

- Identifikation von Adressatenclustern, auf die eine Nachricht in Abhängigkeit eines Objekttyps abgebildet wird.
- Extraktion einer gemeinsamen Schnittstelle für den Adressatencluster
- Anreicherung der Schnittstelle um mögliche ähnliche/identische Implementierungen
- Anpassung der Clients, so dass diese polymorph gegen die Schnittstelle entwickelt werden.

2.1.40 Späte Abstraktion

Definition

Eine abstrakte Klasse hat - direkt oder indirekt - eine nicht-abstrakte Oberklasse.

Beschreibung

Das Mittel der Abstraktion hilft, Gemeinsamkeiten unterschiedlicher spezifischer Artefakte zu extrahieren. Abgebildet auf Vererbungsstrukturen läßt sich daraus eine Erwartungshaltung ableiten, bei der die abstrakteren Klassen Superklassen von spezifischeren Unterklassen sind.
Wird diese Erwartungshaltung verletzt, d.h. es gibt eine abstrakte Klasse, die direkt oder indirekt von einer nicht-abstrakten Klasse erbt, so ergeben sich hieraus große Verständnisschwierigkeiten. Die unterschiedlichen Eigenschaften von abstrakten und nicht-abstrakten Klassen (z.B. die Möglichkeit zur Instanziierung von Objekten) und die damit notwendige besondere Behandlung je nach Klasse behindern die homogene Sicht auf Vererbungsteilbäume, da z.B. allgemeine Forderungen der Co- und Contravarianz verletzt werden (Oberklassen können bei der späten Abstraktion z.B. etwas, was Unterklassen nicht können, wie z.B. Objekte erzeugen).
Formal kann die späte Abstraktion als Verletzung des Liskov Substitution Principle ([Liskov88], [Martin02]) aufgefaßt werden: die konkrete Klasse kann instanziiert werden aber nicht beliebig durch Instanzen ihrer Ableitungen ersetzt werden - die abstrakte Klasse ist nämlich nicht instaziierbar.

Programmiersprache

C++, Java, C#

Betrachtungsebene

Designebene

Informale Erkennung

- Ermittlung aller abstrakten Klassen im System.
- Für jede Klasse überprüfen, ob es eine nicht-abstrakte Oberklasse gibt.
- Jede abstrakte Klasse mit mindestens einer nicht-abstrakten Oberklasse wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: SpaeteAbstraktion.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:

Eine abstrakte Klasse hat - direkt oder indirekt - eine nicht-abstrakte
Oberklasse.

Beschreibung:

Das Mittel der Abstraktion hilft, Gemeinsamkeiten unterschiedlicher spezifischer
Artefakte zu extrahieren. Abgebildet auf Vererbungsstrukturen bedeutet dies eine
Erwartungshaltung, bei der die abstrakteren Klassen als Superklassen von
spezifischeren Unterklassen bestehen. Wird diese Erwartungshaltung verletzt,
d.h. es gibt eine abstrakte Klasse, die direkt oder indirekt von einer nicht-
abstrakten Klasse erbt, so ergeben sich hieraus große Verständnisschwierigkeiten.
Die unterschiedlichen Eigenschaften von abstrakten und nicht-abstrakten Klassen
(z.B. von welchen können Objekte instanziiert werden) und die damit notwendige
besondere Behandlung je nach Klasse behindern zudem die homogene Sicht auf
Vererbungsteilbäume, da z.B. allgemeine Forderungen der Co- und Contravarianz
verletzt werden (Oberklassen können bei der späten Abstraktion z.B. etwas, was
Unterklassen nicht können, wie z.B. Objekte erzeugen. Formal kann die späte
Abstraktion als Verletzung des Liskov Substitution Principle [LW94] aufgefasst
werden: die konkrete Klasse kann instanziiert werden aber nicht beliebig durch
Instanzen ihrer Ableitungen ersetzt werden - die abstrakte Klasse ist nämlich
nicht instaziierbar.

Implementierung:
- look for all system defined classes that have an inheritance relation
- subclass must have at least one abstract memeber and superclass no such members

(OS) Zweite Bedingung gestrichen, da sie nicht zur Beschreibung passt.
*/

SELECT DISTINCT
       sub.fullname as AbstractSubClass,
       super.fullname as NonAbstractSuperClass
FROM
```

```
        TTypes as sub,
        TTypes as super,
        TSourceEntities as seSub,
        TSourceEntities as seSuper,
        TInheritances as inh,
        TMembers as memSub,
        TMembers as memSuper,
        TConstants as constTypeClass
WHERE
        /* all system defined classes that are related */
        constTypeClass.name = 'TYPE_CLASS' and
        /* avoid interfaces and others */
        super.kindOfType = constTypeClass.value and
        inh.superId=super.id and
        inh.classId=sub.id and
        seSub.id = sub.id and
        seSuper.id = super.id and
        seSub.sourceFileId >= 0 and
        seSuper.sourceFileId >= 0 and
        memSub.id = sub.id and
        memSuper.id = super.id and
        memSub.isabstract = 1 and
        memSuper.isabstract = 0
;
```

Lösungsstrategie

Die Behebung des Problems ist sehr vielschichtig. Eine einfache, aber nur in den wenigsten Fällen sinnvolle Restrukturierung bezieht sich auf das Beseitigen der Abstraktheit durch Hinzufügen der Funktionalität, die das Arbeiten mit Objekten dieser Klasse ermöglichen. Dies sollte jedoch nur geschehen, wenn diese Objekte in der Realität auch tatsächlich auftreten. Eine andere Alternative ist das Entfernen der Vererbungsbeziehung der abstrakten Klasse zu ihrer Superklasse. Stattdessen können die Signaturen der abstrakten Klasse in ein Interface ausgelagert werden, das keine weiteren Superklassen besitzt. Die Unterklassen der ehemaligen abstrakten Klasse erben anschließend von der Superklasse der ehemaligen abstrakten Klasse und implementieren das neue Interface. Je nach Kontext kann es auch sinnvoll sein, die gesamte Vererbungskette zwischen der abstrakten Klasse und ihrer Superklassen aufzubrechen. Die abstrakte Klasse bleibt hierbei abstrakt, verwendet die Oberklassen allerdings nur noch via Delegation.

2.1.41 Tote Attribute

Definition

Ein privates Attribut / eine private Konstante einer Klasse wird von dieser an keiner Stelle verwendet.

Beschreibung

Eine Klasse dient der Implementierung eines abstrakten Datentyps, d.h. dem Anbieten von Services auf Basis von klasseninternen Daten. Werden nun Daten intern deklariert, aber für die Bereitstellung der Services nicht benötigt, sind diese überflüssig und vergrößern lediglich die Systemgröße.
Die betrachtete Sichtbarkeit ist auf „private“ beschränkt, da nur in diesem Fall eine spätere (externe) Verwendung mit hoher Wahrscheinlichkeit ausgeschlossen werden kann.
Referenzen:

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

- Selektion aller privaten Attribute und privaten Konstanten.

- Selektion derjenigen privaten Attribute/Konstanten, die keine Verwendung aufweisen.
- Jedes so gefundene Artefakt stellt eine Verletzung dar.

Erkennung mit SISSy

```
/* $Id: ToteAttribute.sql,v 1.2 2006/01/19 13:51:17 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Name: Tote Attribute

Definition:

Ein privates Attribut / eine private Konstante einer Klasse wird von
dieser an keiner Stelle verwendet.

Beschreibung:

Eine Klasse dient der Implementierung eines abstrakten Datentyps, d.h. dem
Anbieten von Services auf Basis von klasseninternen Daten. Werden nun Daten
intern deklariert, aber für die Bereitstellung der Services nicht benötigt, sind
diese überflüssig und vergrößern lediglich die Systemgröße.

Implementierung:

Es werden alle private Attribute ausgegeben, die in TAccesses nicht vorhanden
sind

*/


SELECT DISTINCT
       C.id AS ClassId,
       C.fullName AS ClassFullName,
       a.id As UnusedFieldId,
       a.name As UnusedFieldName
FROM
       TTypes As C join
       TVariables AS v on (v.classId = C.id) join
       TMembers As a on (a.id = v.id) join
       TConstants AS constField  on (v.kindOfVariable = constField.Value) join
       TConstants AS constPrivate on (a.visibility = constPrivate.Value)
WHERE
       /* private attributes of C */
       constField.name = 'VAR_FIELD' AND
       constPrivate.name = 'VISIBILITY_PRIVATE' AND
       a.name <> '<self>'

/* take out those that have accesses */
EXCEPT

SELECT DISTINCT
       C.id AS ClassId,
       C.fullName AS ClassFullName,
       a.id As UnusedFieldId,
       a.name As UnusedFieldName
FROM
       TTypes As C join
       TVariables AS v on (v.classId = C.id) join
       TMembers As a on (a.id = v.id) join
       TConstants AS constField  on (v.kindOfVariable = constField.Value) join
       TConstants AS constPrivate on (a.visibility = constPrivate.Value) join
       TAccesses AS acc on ( acc.targetId = v.id) join
       TFunctions AS accBelongsToFunction on (acc.functionid = accBelongsToFunction.id) join
       TConstants AS funcType on (funcType.value = accBelongsToFunction.kindoffunction)
WHERE
       /* private attributes of C */
       constField.name = 'VAR_FIELD' AND
       constPrivate.name = 'VISIBILITY_PRIVATE'

       AND

       /* Not accessed from own initializer */
       ((funcType.name  =  'FUNC_INITIALIZER'  AND  accBelongsToFunction.name  <>  v.name)  OR
funcType.name <> 'FUNC_INITIALIZER')

;
```

Lösungsstrategie

Die Behebung des Problems besteht lediglich aus dem direkten Entfernen des toten Attributs

2.1.42 Tote Implementierung

Definition

Eine tote Implementierung liegt vor, wenn Anweisungen existieren, die bei keinem möglichen Kontrollfluss durchlaufen und damit unter keinen Umständen ausgeführt werden.

Beschreibung

Sämtlicher Code sollte minimal sein, d.h. jede einzelne Anweisung dient direkt oder indirekt zur Sicherstellung der Qualität (funktional als auch nichtfunktional) der Anwendung. Existieren Anweisungen, die unter keinen Umständen durchlaufen werden, sind sie lediglich überflüssig und erschweren damit sämtliche Wartungsarbeiten am Code.
Toter Code, der zuverlässig mit Werkzeugen identifiziert werden kann, tritt auf

- nach einer "return"- Anweisung
- nach dem Werfen einer Exception.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

- Ermitteln aller return- und throw-Anweisungen
- Als Verletzung werden alle return und throw-Anweisungen gezählt, auf die unmittelbar Anweisungen folgen, die aufgrund des durch return oder throw unterbrochenen Kontrollflusses niemals ausgeführt werden.

Erkennung mit SISSy

```
/* $Id: ToteImplementierung.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:

Anweisungen, die bei keinem möglichen Kontrollfluss durchlaufen werden und damit
unter keinen Umständen ausgeführt werden.

Beschreibung:

Sämtlicher Code sollte minimal sein, d.h. jede einzelne Anweisung dient direkt
oder indirekt zur Sicherstellung der Qualität (funktional als auch
nichtfunktional) der Anwendung. Existieren Anweisungen, die unter keinen
Umständen durchlaufen werden, sind sie lediglich überflüssig und erschweren
damit sämtliche Wartungsarbeiten am Code. Toter Code, der zuverlässig mit
Werkzeugen identifiziert werden kann, tritt auf
    * nach einer "return"- Anweisung
    * nach dem Werfen einer Exception.

Implementierung:
*/
SELECT DISTINCT
       exitStatement.id as statementid,
       type.name as classname,
       func.name as functioncontainingdeadcode,
       con.name as deadcodeafter,
       files.pathname as filename
FROM
```

```
        TStatements exitStatement join
        TStatements      nextStatement      on      (nextStatement.parentStatementid      =
exitStatement.parentStatementid) join
        TStatements parentStatement on (exitStatement.parentStatementid = parentStatement.id)
join
        TConstants con2 on (parentStatement.kindofstatement = con2.value) join
        TConstants con on (exitStatement.kindofstatement = con.value) join
        TFunctions func on (exitStatement.functionid = func.id) join
        TTypes type on (func.classid = type.id) join
        TSourceEntities sourceEntities on (type.id = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id)

WHERE
        (con.name = 'STATEMENT_RETURN' OR
        con.name = 'STATEMENT_THROW' ) AND
        nextStatement.position > exitStatement.position AND
        con2.name = 'STATEMENT_BLOCK' AND
        type.name <> ''
```

Lösungsstrategie

Die Behebung des Problems besteht lediglich aus dem direkten Entfernen des toten Codes.

2.1.43 Tote Methoden

Definition

Eine private Methode einer Klasse wird von dieser an keiner Stelle aufgerufen.

Beschreibung

Mittels privater Methoden ist es möglich, höherwertige Services, die von der Klasse nach außen gereicht werden, zu implementieren. Hierfür werden die privaten Methoden von den Services direkt oder indirekt über andere Methoden aufgerufen.
Private Methoden, die weder von Services, noch von anderen Methoden verwendet werden, sind daher überflüssig und vergrößern lediglich den Umfang der Klasse.
Die betrachtete Sichtbarkeit ist auf „private“ beschränkt, da nur in diesem Fall eine spätere (externe) Verwendung mit hoher Wahrscheinlichkeit ausgeschlossen werden kann.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Identifikation aller privaten Methode.
- Selektion derjenigen privaten Methoden, die niemals aufgerufen werden.
- Jede so gefundene Methode wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: ToteMethode.sql,v 1.2 2006/01/19 13:51:17 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:

Eine private Methode einer Klasse wird von dieser an keiner Stelle aufgerufen.

Beschreibung:

Mittels privater Methoden ist es möglich, höherwertige Services, die von der
```

```
Klasse nach außen gereicht werden, zu implementieren. Hierfür werden die
privaten Methoden von den Services direkt oder indirekt über andere private
Methoden aufgerufen. Private Methoden, die weder von Services, noch von anderen
Methoden verwendet werden, sind daher überflüssig und vergrößern lediglich den
Umfang der Klasse.

Implementierung:
- select all private methods that are not target to any FUNCACCESS
*/

SELECT DISTINCT
        C.fullName AS ClassFullName,
        m.name As DeadMethodName
FROM
        TTypes As C,
        TMembers AS m,
        TFunctions AS f,
        TConstants AS constPrivate
WHERE
        /* private methods of C */
        f.id = m.id and
        m.classId = C.id and
        m.visibility = constPrivate.Value AND
        constPrivate.name = 'VISIBILITY_PRIVATE'

/* take out those that have accesses */
EXCEPT

SELECT DISTINCT
        C.fullName AS ClassFullName,
        m.name As DeadMethodName
FROM
        TTypes As C,
        TMembers AS m,
        TFunctions AS f,
        TConstants AS constPrivate,
        TAccesses AS acc
WHERE
        /* private methods of C */
        f.id = m.id and
        m.classId = C.id and
        m.visibility = constPrivate.Value AND
        constPrivate.name = 'VISIBILITY_PRIVATE' AND

        /* there is at least an access */
        acc.targetId = m.id AND
        acc.functionid <> f.id
;
```

Lösungsstrategie

Die Behebung des Problems besteht lediglich aus dem direkten Entfernen der toten Methode.

2.1.44 Überbuchte Datei

Definition

Eine Überbuchung liegt vor, wenn in einer logischen Datei mehrere nicht-geschachtelte Klassen mit der Sichtbarkeit public oder default deklariert sind.

Beschreibung

In den betrachteten Programmiersprachen wird der Code immer in Dateien abgelegt. Die grundsätzlich gegebene Möglichkeit, innerhalb einer Datei mehrere Klassen abzulegen, erschwert deutlich das Wiederfinden von Klassen sowie das selektive Verwenden einzelner Klassen aus derart überbuchten Dateien. Idealerweise findet sich daher pro Datei auch nur eine öffentliche bzw. paketsichtbare Klasse.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Technologieebene

Informale Erkennung

- Pro Datei werden die nicht-geschachtelten Klassen mit public oder default Sichtbarkeit gezählt.
- Jede Datei mit mehr als einer solchen Klasse wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: UeberbuchteDatei.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:

Eine Überbuchung liegt vor, wenn in einer logischen Datei mehrere nicht-private,
nicht-geschachtelte Klassen deklariert sind.

Beschreibung:

In den betrachteten Programmiersprachen wird der Code immer in Dateien abgelegt.
Während in JAVA zwischen dem Inhalt und der umgebenden Datei eine 1:1-Abbildung
erzwungen wird, gilt dies für C, C++, C# und Delphi nicht. Die grundsätzlich
gegebene Möglichkeit, innerhalb einer Datei mehrere Klassen abzulegen erschwert
allerdings deutlich das Wiederfinden von Klassen sowie das selektive Verwenden
einzelner Klassen aus derart überbuchten Dateien. Idealerweise entspricht der
Klassenname dem Dateinamen.

Implementierung:
- select files that have more than one public or package class
- these classes are not inner classes

(OS)
- Neu implementiert, da alte Anfrage keine Treffer liefert

*/


create or replace view VPublicOrPackageClasses as

select distinct
       C.id as ClassId
from
       TTypes as C join
       TMembers as meC on (meC.id = C.id) join
       TConstants as con on (meC.visibility = con.value)
where
       C.classid = -1 AND
       (con.name = 'VISIBILITY_PUBLIC'
       OR
       con.name = 'VISIBILITY_PACKAGE')
;


SELECT
       seClass.sourceFileId as FileID,
       meFile.pathname as FileName,
       COUNT(*) as NumberOfClasses
FROM
       VPublicOrPackageClasses as PPClasses inner join
       TSourceEntities as seClass on seClass.id = PPClasses.ClassId inner join
       TFiles as meFile on meFile.id = seClass.sourceFileId
GROUP BY
       seClass.sourceFileId,
       meFile.pathname
HAVING
       COUNT(*) > 1
ORDER BY 3 DESC
;



drop view VPublicOrPackageClasses;
```

Lösungsstrategie

Die Behebung einer überbuchten Dateien geschieht in drei Schritten:

- Ermittlung der "Hauptklasse" einer Datei. Ggfs. namentliche Anpassung der Datei bzw. Hauptklasse
- Für jede weitere Klasse in der überbuchten Datei: Überprüfen, ob die Klasse bzgl. der Sichtbarkeit oder des Scopes eingeschränkt werden kann oder ob eine neue Datei für die jeweilige Klasse angelegt werden muss.
- Ggfs. entfernen der Klasse und Auslagern in die neue Datei.

2.1.45 Unfertiger Code

Definition

Im Quelltext eines Systems existieren Kommentare, die Hinweise darauf geben, daß der Code noch nicht fertig ist. Diese Hinweise sind gegeben durch Vorkommen der Worte „todo", "hack" und "fixme", unabhängig ihrer Groß- und Kleinschreibung.

Beschreibung

Im Tagesgeschäft kann es aus unterschiedlichen Gründen immer wieder notwendig werden, kurzfristig Workarounds in das System zu implementieren. Diese Workaround, die deutlich suboptimale Lösungen darstellen, sollten von einem Entwickler in jedem Fall entsprechend kommentiert werden. Eine hervorragende Möglichkeit ist z.B. die Verwendung formaler Dokumentationstechniken wie JavaDoc oder DoxyGen. Dort sollte ein entsprechendes Tag etabliert werden (z.B. @todo).
Im produktiven Code dagegen sollten solche Workarounds nicht mehr existieren, da dann offensichtlich die kurzfristig notwendig gewordenen Flicken in den langfristig gedachten Produktivcode überführt wurden.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Informale Erkennung

- Gesucht werden die in der Definition aufgeführten Schlüsselwörter „todo“, „hack“ und „fixme“, unabhängig von ihrer Groß-/Kleinschreibung.
- Jeder Kommentarblock, der mindestens eines der genannten Schlüsselwörter enthält, wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: UnfertigerCode.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:
Im Quelltext eines Systems existieren Kommentare, die Hinweise darauf geben, daß
der Code noch nicht fertig ist. Diese Hinweise sind gegeben durch Vorkommen der
Strings "todo", "hack" und "fixme", unabhängig ihrer Groß- und Kleinschreibung.

Beschreibung:
Im Tagesgeschäft kann es aus unterschiedlichen Gründen immer wieder notwendig
werden, kurzfristig Workarounds in das System zu implementieren. Diese
Workaround, die deutlich suboptimale Lösungen darstellen, sollten von einem
Entwickler in jedem Fall entsprechend kommentiert werden. Eine hervorragende
Möglichkeit ist z.B. mittels formaler Dokumentationstechniken wie JavaDoc oder
DoxyGen. Dort sollte ein entsprechendes Tag etabliert werden (z.B. @todo). Im
produktiven Code dagegen sollten solche Workarounds nicht mehr existieren, da
dann offensichtlich die kurzfristig notwendig gewordenen Flicken in den
```

```
langfristig gedachten Produktivcode überführt wurden.

Implementierung:
Erkennung bei gegenwärtigem Werkzeugstand nicht möglich, da Kommentare nicht
gespeichert werden.
*/

/* Prints the places, where you can find the todos in sourcecode. Useful, if you would like to
get rid of them */
/*
SELECT
        'TODO',
        cSourceEntity.startline,
        cSourceEntity.sourcefileid,
        cfile.pathname


FROM
        TComments comments join
        TSourceEntities cSourceEntity on (comments.id = cSourceEntity.id)  join
        TFiles cfile on (cSourceEntity.sourcefileid = cfile.id)
WHERE
        comments.numberoftodos > 0

;
*/


/* Prints the number of comments containing a todo */
SELECT
        count(*)
FROM
        TComments comments join
        TSourceEntities cSourceEntity on (comments.id = cSourceEntity.id)  join
        TFiles cfile on (cSourceEntity.sourcefileid = cfile.id)
WHERE
        comments.numberoftodos > 0
```

Lösungsstrategien

Für die Behebung kann kein generelles Vorgehen angegeben werden. Die Behebung des Problems erfordert eben die Implementierung fehlender Codebestandteile. Nach der Überarbeitung ist daran zu denken, die Kommentare anzupassen.

2.1.46 Unvollständige Vererbung (Attribut)

Definition

Ein beliebiges, nicht-statisches Attribut existiert namensgleich entweder

- in mehr als 50% der direkten Unterklassen UK (mind. 2) oder
- in mind. 10 der direkten Unterklassen UK

einer Oberklasse OK, aber in keiner direkten Oberklasse der Unterklassen UK.

Beschreibung

Eine mächtige Möglichkeit der Vererbung ist, gemeinsames Verhalten und gemeinsame Zustandsmengen von Unterklassen in einer gemeinsamen Oberklasse herauszufaktorisieren. Trotz der mehrfachen Verwendung in den Unterklassen muss das Verhalten dann nur noch einmal verstanden, getestet und gewartet werden. Da Vererbungsstrukturen allerdings starken Veränderungen unterliegen, kommt es insbesondere bei "älteren" Vererbungsstrukturen dazu, dass derartige Faktorisierungspotentiale ungenutzt bleiben. Die Folge sind redundante Implementierungen mit mehrfachen Aufwänden für Testen, Warten und Verstehen.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Suchen nach Attributen mit gleichem Namen in direkten Unterklassen UK einer gemeinsamen Oberklasse OK.
- Entfernung derjenigen Attributnamen aus der im vorherigen Schritt ermittelten Menge, die in einer beliebigen (in-)direkten Oberklasse deklariert sind (da dies eine Attributüberdeckung wäre).
- Filterung entsprechend der Grenzwerte für Unterklassen UK (mind 2, >50%) bzw. (mind.10)
- Jeder Attributname einer maximalen Unterklassenmenge der obige Bedingungen erfüllt, wird als eine Verletzung gezählt

Erkennung mit SISSy

```
/* $Id: UnvollstaendigeVererbungAttribut.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:
Ein beliebiges, nicht-statisches Attribut existiert namensgleich entweder
    * in mehr als 50% der direkten Unterklassen UK (mind. 2) oder
    * in mind. 10 der direkten Unterklassen UK
einer Oberklasse OK, aber in keiner direkten Oberklasse der Unterklassen UK.

Beschreibung:
Eine mächtige Möglichkeit der Vererbung ist, gemeinsames Verhalten und
gemeinsame Zustandsmengen von Unterklassen in eine gemeinsame Oberklasse
herauszufaktorisieren. Trotz der mehrfachen Verwendung in den Unterklassen muss
das Verhalten dann nur noch einmal verstanden, getestet und gewartet werden. Da
Vererbungsstrukturen allerdings starken Veränderungen unterliegen, kommt es
insbesondere bei "älteren" Vererbungsstrukturen dazu, dass derartige
Faktorisierungspotentiale ungenutzt bleiben. Die Folge sind redundante
Implementierungen mit mehrfachen Aufwänden für Testen, Warten und Verstehen.


*/



create or replace view VAttributeInSubClasses as

select
        variable.name as variablename,
        supertype.id as supertypeid,
        supertype.name as supertypename,
        count (supertype.id)
from
        TVariables variable join
        TMembers vmember on (variable.id = vmember.id) join
        TTypes type on ( variable.classid = type.id) join
        TInheritances inh on (inh.classid = type.id) join
        TTypes supertype on (inh.superid = supertype.id) join
        TConstants con on ( con.value = variable.kindofvariable) join
        TSourceEntities source on (source.id = supertype.id) join
        TFiles file on (source.sourcefileid = file.id)
where
        con.name = 'VAR_FIELD' and
        inh.depthofinheritance = 1 and
        variable.name <> '<self>' and
        vmember.isstatic = 0 and

        not exists (
                select
                        *
                from
                        TVariables variable2
                where
                        variable2.classid = supertype.id and
                        variable2.name = variable.name
        )


group by
        variable.name,
```

```
        supertype.id,
        supertype.name

;


create or replace view VNumberOfSubclasses as
select
        type.id,
        type.name,
        count (inh.classid)
from
        TTypes type join
        TInheritances inh on (inh.superid = type.id)
where
        inh.depthofinheritance = 1
group by
        type.id,
        type.name
;

select
        *
from
        VAttributeInSubClasses aisc join
        VNumberOfSubclasses nosc on (aisc.supertypeid = nosc.id)
where
        aisc.count > 1 and (
        aisc.count > 10 or
        (aisc.count * 100 / nosc.count * 100 > 50))
;



drop view VNumberOfSubclasses;
drop view VAttributeInSubClasses;
```

Lösungsstrategie

Die Behebung des Problems geschieht entlang der folgenden Schritte:

- Alle Klassen identifizieren, die ein Attribut gemeinsam haben.
- Herausziehen des Attributes in eine gemeinsame Superklasse. Diese kann entweder die gemeinsame Oberklasse sein, oder wenn nicht alle Unterklassen einer Superklasse das Attribut haben, eine neue Abstraktion sein.

2.1.47 Unvollständige Vererbung (Methode)

Definition

Eine beliebige Methodensignatur existiert entweder

- in mehr als 50% der direkten Unterklassen UK (mind. 2) oder
- in mind. 10 der direkten Unterklassen UK

einer Oberklasse OK, die in keiner direkten Oberklasse der Unterklassen UK bekannt ist. Als Klassen (OK, UK) werden in JAVA auch Interfaces verstanden.

Beschreibung

Eine mächtige Möglichkeit der Vererbung ist, gemeinsames Verhalten von Unterklassen in eine gemeinsame Oberklasse herauszufaktorisieren. Trotz der mehrfachen Verwendung in den Unterklassen muss das Verhalten dann nur noch einmal verstanden, getestet und gewartet werden. Da Vererbungsstrukturen allerdings starken Veränderungen unterliegen, kommt es insbesondere bei "älteren" Vererbungsstrukturen dazu, dass derartige Faktorisierungspotentiale ungenutzt bleiben. Die Folge sind redundante Implementierungen mit mehrfachen Aufwänden für Testen, Warten und Verstehen.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Suchen nach Methoden mit gleichen Signaturen in direkten Unterklassen UK einer gemeinsamen Oberklasse OK.
- Entfernung derjenigen Methodensignaturen aus der im vorherigen Schritt ermittelten Menge, die in einer beliebigen (in-)direkten Oberklasse deklariert sind.
- Filterung entsprechend der Grenzwerte für Unterklassen UK (mind 2, >50%) bzw. (mind.10)
- Jede so ermittelte Methodensignatur einer maximalen Unterklassenmenge wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: UnvollstaendigeVererbungMethode.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:
Eine beliebige Methodensignatur existiert entweder
    * in mehr als 50% der direkten Unterklassen UK (mind. 2) oder
    * in mind. 10 der direkten Unterklassen UK
einer Oberklasse OK, die in keiner direkten Oberklasse der Unterklassen UK
bekannt ist.

Beschreibung:
Eine mächtige Möglichkeit der Vererbung ist, gemeinsames Verhalten von
Unterklassen in eine gemeinsame Oberklasse herauszufaktorisieren. Trotz der
mehrfachen Verwendung in den Unterklassen muss das Verhalten dann nur noch
einmal verstanden, getestet und gewartet werden. Da Vererbungsstrukturen
allerdings starken Veränderungen unterliegen, kommt es insbesondere bei
"älteren" Vererbungsstrukturen dazu, dass derartige Faktorisierungspotentiale
ungenutzt bleiben. Die Folge sind redundante Implementierungen mit mehrfachen
Aufwänden für Testen, Warten und Verstehen.

Implementierung:
-Sucht nach Am--Bm Methodenpaaren, die die selbe Signatur haben aber nicht
geerbt werden.
*/


/* $Id: UnvollstaendigeVererbungMethode.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:
Ein beliebiges, nicht-statisches Attribut existiert namensgleich entweder
    * in mehr als 50% der direkten Unterklassen UK (mind. 2) oder
    * in mind. 10 der direkten Unterklassen UK
einer Oberklasse OK, aber in keiner direkten Oberklasse der Unterklassen UK.

Beschreibung:
Eine mächtige Möglichkeit der Vererbung ist, gemeinsames Verhalten und
gemeinsame Zustandsmengen von Unterklassen in eine gemeinsame Oberklasse
herauszufaktorisieren. Trotz der mehrfachen Verwendung in den Unterklassen muss
das Verhalten dann nur noch einmal verstanden, getestet und gewartet werden. Da
Vererbungsstrukturen allerdings starken Veränderungen unterliegen, kommt es
insbesondere bei "älteren" Vererbungsstrukturen dazu, dass derartige
Faktorisierungspotentiale ungenutzt bleiben. Die Folge sind redundante
Implementierungen mit mehrfachen Aufwänden für Testen, Warten und Verstehen.


*/



create or replace view VMethodsInSubClasses as

select
```

```
        func.name as functionname,
        sig.signature as signature,
        supertype.id as supertypeid,
        supertype.name as supertypename,
        count (supertype.id)
from
        TFunctions func join
        TConstants con on (func.kindoffunction = con.value) join
        TSignatures sig on (sig.functionid = func.id) join
        TTypes type on ( func.classid = type.id) join
        TInheritances inh on (inh.classid = type.id) join
        TTypes supertype on (inh.superid = supertype.id) join
        TSourceEntities source on (source.id = supertype.id) join
        TFiles file on (source.sourcefileid = file.id)
where
        con.name = 'FUNC_METHOD' and
        inh.depthofinheritance = 1 and

        not exists (
                select
                        *
                from
                        TSignatures sig2 join
                        TFunctions func2 on (sig2.functionid = func2.id) join
                        TInheritances inh2 on (inh2.classid = supertype.id) join
                        TInheritances inh3 on (inh3.classid = type.id)
                where
                        sig2.signature = sig.signature and
                        (func2.classid = supertype.id or
                         func2.classid = inh2.superid or
                         func2.classid = inh3.superid)
        )


group by
        func.name,
        sig.signature,
        supertype.id,
        supertype.name

;


create or replace view VNumberOfSubclasses as
select
        type.id,
        type.name,
        count (inh.classid)
from
        TTypes type join
        TInheritances inh on (inh.superid = type.id)
where
        inh.depthofinheritance = 1
group by
        type.id,
        type.name
;

select
        *
from
        VMethodsInSubClasses misc join
        VNumberOfSubclasses nosc on (misc.supertypeid = nosc.id)
where
        misc.count > 1 and (
        misc.count > 10 or
        (misc.count * 100 / nosc.count * 100 > 50))
;



drop view VNumberOfSubclasses;
drop view VMethodsInSubClasses;
```

Lösungsstrategie

Die Behebung des Problems geschieht entlang der folgenden Schritte:

- Alle Klassen identifizieren, die eine Methode gemeinsam haben.
- Überprüfen, inwieweit diese Methoden identisch implementiert sind.
- Herausziehen der Methode in eine gemeinsame Abstraktion. Diese kann entweder die bestehende Oberklasse oder eine neue Unterklasse dieser sein. Wenn die Implementierung in den Unterklassen identisch ist, kann die Methode mit der Implementierung herausgezogen werden. Ansonsten genügt die Signatur. Das Herausziehen kann, wenn nicht alle Unterklassen einer Superklasse diese Methode haben, auch als eine neue Schnittstelle passieren, die die entsprechenden Unterklassen implementieren müssen.

2.1.48 Verbotene Dateiliebe

Definition

Zwischen zwei logischen Dateien besteht eine zyklische Abhängigkeit, da die Inhalte der logischen Dateien sich jeweils gegenseitig für das Compilieren benötigen.

Beschreibung

Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten, die jeweils separat verstehbar, testbar und wartbar sind. Abhängigkeiten zu anderen Teilen, die zwangsläufig entstehen, um ein Gesamtsystem aufbauen zu können, werden i.d.R. in Form von Schichten aufgebaut, d.h. es gibt reine Dienstleister (z.B. I/O-Funktionalität), die von der nächst höherwertigen Schicht verwendet werden, die ihrerseits wieder Funktionalität einer noch höheren Schicht anbietet. Diese Modularität sollte sich insbesondere auf Dateiebene wiederfinden, um zu vermeiden, dass zwei Dateien jeweils nur zusammen betrachtbar sind. Anderenfalls benötigen die Inhalte der einen Datei die der anderen und umgekehrt. Liegt eine solche gegenseitige Abhängigkeit vor, müssen beide Dateien grundsätzlich zusammen betrachtet werden.

Programmiersprache

C++, Java, C#, ~~Delphi~~

Betrachtungsebene

Codeebene

Informale Erkennung

- Ermitteln aller logischen Dateipaare, die gegenseitig Inhalte verwenden.
- Jedes Paar wird als eine Verletzung gezählt.

Erkennung mit SISSy

```
/* $Id: VerboteneDateiliebe.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:
Zwischen zwei logischen Dateien besteht eine zyklische Abhängigkeit, da die
Inhalte der logischen Dateien sich jeweils gegenseitig für das Compilieren
benötigen.

Beschreibung:
Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten,
die jeweils separat verstehbar, testbar und wartbar sind. Abhängigkeiten zu
anderen Teilen, die zwangsläufig entstehen, um ein Gesamtsystem aufbauen zu
können, werden i.d.R. in Form von Schichten aufgebaut, d.h. es gibt reine
Dienstleister (z.B. I/O-Funktionalität), die auf der nächst höherwertigen
Schicht verwendet werden, die ihrerseits wieder Funktionalität einer noch
höheren Schicht anbietet. Diese Modularität sollte sich insbesondere auf
Dateiebene wiederfinden. Bei der verbotenen Dateiliebe dagegen sind die
Abhängigkeiten, die ausschließlich auf Basis der Include- bzw. Import-
Abhängigkeit ermittelt werden, so gewählt, dass zwei Dateien jeweils nur
zusammen betrachtbar sind, d.h. die Inhalt der einen Datei benötigen die Inhalte
der anderen Datei und umgekehrt. Dies hat zur Folge, dass beide Dateien jeweils
nur monolithisch zusammen bearbeitet werden können.
```

```
Implementierung:
look for pairs of files
- access each other's types or global functions
- or their classes inherit from classes of the other
- duplicate pairs must be avoided

Note: optimized for performance (AT)

(OS) OffenderIDs von der Ausgabe ausgeschlossen, da man sonst zu viele Treffer bekommt

*/

CREATE OR REPLACE VIEW VFileImportsFile AS
select
       f.id as fileid,
       f.pathname as filename,
       f2.id as targetfileid,
       f2.pathname as targetfilename
from
       TFiles f join
       TImports imp on (imp.fileid = f.id) join
       TTypes type on (imp.targetid = type.id) join
       TSourceEntities s on (s.id = type.id) join
       TFiles f2 on (s.sourcefileid = f2.id)
where
       f.id <> f2.id

;

select distinct
       F1.id as FileOneID,
       F1.pathname as FileOnePathname,
       /* seF1.id as OffenderSourceEntityFileOneID, */
       F2.id as FileTwoID,
       F2.pathname as FileTwoPathname /*,
       seF2.id as OffenderSourceEntityFileTwoID */
from
       TSourceEntities as seF1,
       TSourceEntities as seF2,
       TSourceEntities as tempInF1,
       TSourceEntities as tempInF2,
       TAccesses as acc12,
       TAccesses as acc21,
       TFiles as F1,
       TFiles as F2
where
       tempInF1.sourceFileId = seF1.sourceFileId and
       tempInF2.sourceFileId = seF2.sourcefileId and

       /* avoid duplicates */
       seF1.sourceFileId < seF2.sourceFileId and

       tempInF1.sourceFileId = F1.id and
       tempInF2.sourceFileId = F2.id and

       /* dependency from 1 to 2 through access */
       acc12.sourceId = seF1.id and
       acc12.targetId = tempInF2.id and
       /* dependency from 2 to 1 through access */
       acc21.sourceId = seF2.id and
       acc21.targetId = tempInF1.id

union

select distinct
       F1.id as FileOneID,
       F1.pathname as FileOnePathname,
       /*seF1.id as OffenderSourceEntityFileOneID, */
       F2.id as FileTwoID,
       F2.pathname as FileTwoPathname /*,
       seF2.id as OffenderSourceEntityFileTwoID*/
from
       TSourceEntities as seF1,
       TSourceEntities as seF2,
       TInheritances as inh,
       TFiles as F1,
       TFiles as F2
where
       seF1.sourceFileId = F1.id and
```

```
        seF2.sourceFileId = F2.id and

        /* dependency from 1 to 2 through inheritance */
        inh.classId = seF1.id and inh.superId = seF2.id and
        /* dependency from 2 to 1 through inheritance */
        inh.classId = seF2.id and inh.superId = seF1.id      and

        /* avoid duplicates */
        seF1.sourceFileId < seF2.sourceFileId

union

select distinct
        F1.id as FileOneID,
        F1.pathname as FileOnePathname,
        /*seF1.id as OffenderSourceEntityFileOneID, */
        F2.id as FileTwoID,
        F2.pathname as FileTwoPathname /*,
        seF2.id as OffenderSourceEntityFileTwoID */
from
        TSourceEntities as seF1,
        TSourceEntities as seF2,
        TSourceEntities as tempInF2,
        TAccesses as acc12,
        TInheritances as inh,
        TFiles as F1,
        TFiles as F2
where
        tempInF2.sourceFileId = seF2.sourcefileId and

        /* avoid duplicates */
        seF1.sourceFileId < seF2.sourceFileId and

        tempInF2.sourceFileId = F2.id and

        /* dependency from 2 to 1 through inheritance */
        inh.classId = seF2.id and inh.superId = seF1.id       and

        /* dependency from 1 to 2 through access */
        acc12.sourceId = seF1.id and
        acc12.targetId = tempInF2.id

union

select distinct
        F1.id as FileOneID,
        F1.pathname as FileOnePathname,
        /* seF1.id as OffenderSourceEntityFileOneID,*/
        F2.id as FileTwoID,
        F2.pathname as FileTwoPathname
        /* seF2.id as OffenderSourceEntityFileTwoID */
from
        TSourceEntities as seF1,
        TSourceEntities as seF2,
        TSourceEntities as tempInF1,
        TAccesses as acc21,
        TInheritances as inh,
        TFiles as F1,
        TFiles as F2
where
        tempInF1.sourceFileId = seF1.sourceFileId and

        /* avoid duplicates */
        seF1.sourceFileId < seF2.sourceFileId and

        tempInF1.sourceFileId = F1.id and

        /* dependency from 1 to 2 through inheritance */
        inh.classId = seF1.id and inh.superId = seF2.id and
        /* dependency from 2 to 1 through access */
        acc21.sourceId = seF2.id and
        acc21.targetId = tempInF1.id

union

SELECT
        v.fileid,
        v.targetfilename,
        v2.fileid,
        v2.targetfilename
```

```
FROM
       VFileImportsFile v,
       VFileImportsFile v2
WHERE
       v.fileid = v2.targetfileid AND
       v.targetfileid = v2.fileid AND
       v.fileid < v2.fileid

;


DROP VIEW VFileImportsFile;
```

Lösungsstrategie

Dieses Problem entsteht als direkte Folge einer verbotenen Klassen- oder Methodenliebe. Die Behebung erfolgt durch die Anwendung der entsprechenden Behandlungsstrategie.

2.1.49 Verbotene Klassenliebe

Definition

Es gibt eine direkte zyklische Compile-Abhängigkeit zwischen zwei nicht ineinander geschachtelten, nicht in einer (in-)direkten Vererbungsbeziehung stehenden Klassen bzw. Interfaces, die durch die Inhalte der Klassen/Interfaces (z.B. Methodenaufrufe und Signaturdeklarationen) begründet ist.
Eine Compile-Abhängigkeit liegt vor, wenn sich die eine Klasse/Interface nicht ohne die andere compilieren läßt. In C++ muß zusätzlich die Möglichkeit des Linkens gegeben sein (da eine reine `extern`-Deklaration vom Compiler nicht überprüft wird).

Beschreibung

Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten, die jeweils separat verstehbar, testbar und wartbar sind. Dies gilt insbesondere für Klassen/Interfaces. Bei der verbotenen Klassenliebe dagegen sind die Abhängigkeiten, die ausgehend von allen möglichen Abhängigkeitstypen ermittelt werden (z.B. Typzugriff, Methodenaufrufe, Attribut-/Konstantenverwendung usw.) so gerichtet, dass zwei Klassen/Interfaces gegenseitig voneinander abhängig sind, d.h. zum Compilieren der einen Klasse/Interface wird die andere Klasse/Interface benötigt und umgekehrt. Dies hat zur Folge, dass beide Klassen/Interfaces jeweils nur monolithisch zusammen bearbeitet werden können.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Das Problem kann identifiziert werden, indem nicht ineinander geschachtelte Klassenpaare ermittelt werden, die sich gegenseitig über die jeweiligen Inhalte referenzieren.
- Jedes so ermittelte Klassenpaar stellt eine Verletzung dar.

Erkennung mit SISSy

```
/* $Id: VerboteneKlassenLiebe.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:
Es gibt eine direkte zyklische Abhängigkeit zwischen zwei nicht ineinander
geschachtelten, nicht in einer (in-)direkten Vererbungsbeziehung stehenden
Klassen, die durch die Inhalte der Klassen (z.B. Methodenaufrufe) begründet ist.
```

```
Beschreibung:
Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten,
die jeweils separat verstehbar, testbar und wartbar sind. Dies gilt insbesondere
für Klassen. Bei der verbotenen Klassenliebe dagegen sind die Abhängigkeiten,
die ausgehend von allen möglichen Abhängigkeitstypen ermittelt werden (z.B.
Typzugriff, Methodenaufrufe, Attribut-/Konstantenverwendung usw.) so gerichtet,
dass zwei Klassen gegenseitig voneinander abhängig sind, d.h. zur
Implementierung der einen Klasse wird die andere Klasse benötigt und umgekehrt.
Dies hat zur Folge, dass beide Klassen jeweils nur monolithisch zusammen
bearbeitet werden können.

*/

create table VKnowsRel as

select distinct res.classId, res.className,
       res.knownClassId, res.knownClassName
from (

/* Inheritance relations */
select distinct       cls.id as ClassId, cls.fullName as ClassName,
       knowncls.id as KnownClassId, knowncls.fullName as KnownClassName
from   TConstants con, TAccesses acc,
       TTypes cls, TTypes knowncls
where  con.name = 'TYPEACCESS_INHERITANCE' and
       con.value = acc.kindOfAccess and
       acc.sourceId = cls.id and
       acc.targetId = knowncls.id

union

/* Exception declarations and casts */
select distinct       cls.id as ClassId, cls.fullName as ClassName,
       knowncls.id as KnownClassId, knowncls.fullName as KnownClassName
from   TConstants con, TAccesses acc, TTypes cls,
       TMembers m, TTypes knowncls
where  (con.name = 'TYPEACCESS_THROW' or
               con.name = 'TYPEACCESS_CAST') and
       con.value = acc.kindOfAccess and
       acc.functionId = m.id and
       m.classId = cls.id and
       acc.targetId = knowncls.id

union

/* Attributes and properties */
select distinct       cls.id as ClassId, cls.fullName as ClassName,
       knowncls.id as KnownClassId, knowncls.fullName as KnownClassName
from   TTypes as cls, TMembers as m,
       TVariables as var, TAccesses as acc,
       TTypes knowncls
where  cls.id = m.classId and
       m.id = var.id and
       var.typeDeclarationId = acc.id and
       acc.targetId = knowncls.id

union

/* Formal parameters, catch parameters and local variables */
select distinct       cls.id as ClassId, cls.fullName as ClassName,
       knowncls.id as KnownClassId, knowncls.fullName as KnownClassName
from   TTypes as cls, TMembers as m,
       TVariables as var, TAccesses as acc,
       TTypes knowncls
where  cls.id = m.classId and
       m.id = var.functionId and
       var.typeDeclarationId = acc.id and
       acc.targetId = knowncls.id

union

/* Return type */
select distinct       cls.id as ClassId, cls.fullName as ClassName,
       knowncls.id as KnownClassId, knowncls.fullName as KnownClassName
from   TTypes as cls, TMembers as m,
       TFunctions as func, TAccesses as acc,
       TTypes knowncls
where  cls.id = m.classId and
       m.id = func.id and
```

```
        func.returnTypeDeclarationId = acc.id and
        acc.targetId = knowncls.id
union

/* Accesses to members */
select distinct       cls.id as ClassId, cls.fullName as ClassName,
        knowncls.id as KnownClassId, knowncls.fullName as KnownClassName
from    TTypes as cls, TMembers as sm,
        TAccesses acc, TMembers as dm, TTypes knowncls
where   cls.id = sm.classId and
        acc.functionId = sm.id and
        acc.targetId = dm.id and
        dm.classId = knowncls.id
) as res, TTypes as types, TConstants as c
where   res.classId != res.knownClassId and
        res.knownClassId = types.id and
        types.kindOfType = c.value and
        (c.name like '%CLASS' or c.name like '%INTERFACE')
;


SELECT DISTINCT
        class1.id AS class1_ID,
        class1.fullName AS class1_FullName,
        class2.id AS class2_ID,
        class2.fullName AS class2_FullName
FROM
        TTypes class1,
        TTypes class2,
        VKnowsRel vKnowsRel,
        VKnowsRel vKnowsRel2,
        TSourceEntities class1source,
        TFiles  class1file,
        TSourceEntities class2source,
        TFiles class2file
WHERE
        vknowsRel.classId = class1.id AND
        vknowsRel.knownClassId = class2.id AND
        vknowsRel2.classId = class2.id AND
        vknowsRel2.knownClassId = class1.id AND
        /* Nur Source-Pakete analysieren */
        class1source.id = class1.id AND
        class1source.sourcefileid = class1file.id AND
        class2source.id = class2.id AND
        class2source.sourcefileid = class2file.id AND
        class1.id < class2.id

        /* Keine der Klassen ist Unterklasse der Anderen*/
        AND NOT EXISTS (
        SELECT
                inherits.classid
        FROM
                TInheritances inherits
        WHERE
                (inherits.classid = class1.id AND inherits.superid = class2.id) OR
                (inherits.classid = class2.id AND inherits.superid = class1.id)

        UNION

        SELECT
                tcr.classid
        FROM
                TClassContainmentRelations tcr
        WHERE
                (class1.id = tcr.classid      AND     class2.id = tcr.containingclassid)
                OR (class2.id = tcr.classid   AND     class1.id = tcr.containingclassid)
        )


;

drop table vknowsrel;
```

Lösungsstrategie

Es muss unterschieden werden zwischen mehreren möglichen Ursachen für die zyklische Abhängigkeit zwischen Klassen:

- Bei Klasseninzest sollte man die Behebungsstrategie von 2.1.20 anwenden
- Bei gegenseitigem Zugriff auf Attributen oder Methoden ist zuerst zu überprüfen ob eine oder mehrere dieser Attributen/Methoden eventuell falsch platziert worden sind. In einem solchen Fall sollte man die entsprechenden Attributen /Methoden je nach Bedarf, in die eine, die andere oder eine dritte Klasse verschieben.
- Wenn zwischen den Klassen viele Gemeinsamkeiten existieren, könnte es sich lohnen eventuell eine gemeinsame Oberklasse oder Schnittstelle zu definieren, die dann beide referenzieren (siehe auch 2.1.46 und 2.1.47).

2.1.50 Verbotene Methodenliebe

Definition

Zwei unterschiedliche Methoden rufen sich gegenseitig direkt, nicht-polymorph auf.

Beschreibung

Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten, die jeweils separat verstehbar, testbar und wartbar sind. Dies gilt auf Datei-, Klassen- als auch auf Methodenebene. Abhängigkeiten zu anderen Teilen, die zwangsläufig entstehen, um ein Gesamtsystem aufbauen zu können, werden i.d.R. in Form von Schichten aufgebaut, d.h. es gibt reine Dienstleister (z.B. I/O-Funktionalität), die von der nächst höherwertigen Schicht verwendet werden, die ihrerseits wieder Funktionalität einer noch höheren Schicht anbietet. Auf Methodenebene wird diese Schichtenarchitektur u.a. durch die verwendeten Sichtbarkeiten ausgedrückt (private Methoden sind Basis-Dienstleister, die den höherwertigen Services für ihre Realisierung dienen).
Bei der verbotenen Methodenliebe dagegen sind die Abhängigkeiten auf Basis der direkten Verwendung so gewählt, dass zwei Methoden jeweils nur zusammen betrachtbar sind, d.h. die Inhalte der einen Methode benötigen die Inhalte der anderen Methode und umgekehrt. Dies hat zur Folge, dass beide Methoden jeweils nur monolithisch zusammen bearbeitet werden können.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Informale Erkennung

- Ermitteln aller Paare von unterschiedlichen Methoden, die sich gegenseitig aufrufen.
- Jedes so ermittelte Methodenpaar stellt eine Instanz des Problemmusters dar.

Erkennung mit SISSy

```
/* $Id: VerboteneMethodenLiebe.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:
Zwei unterschiedliche Methoden rufen sich gegenseitig direkt, nicht-polymorph
auf.

Beschreibung:
Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten,
die jeweils separat verstehbar, testbar und wartbar sind. Dies gilt auf Datei-,
Klassen- als auch auf Methodenebene. Abhängigkeiten zu anderen Teilen, die
zwangsläufig entstehen, um ein Gesamtsystem aufbauen zu können, werden i.d.R. in
Form von Schichten aufgebaut, d.h. es gibt reine Dienstleister (z.B.
I/O-Funktionalität), die auf der nächst höherwertigen Schicht verwendet werden,
die ihrerseits wieder Funktionalität einer noch höheren Schicht anbietet. Auf
Methodenebene wird diese Schichtenarchitektur u.a. durch die verwendeten
Sichtbarkeiten ausgedrückt (private Methoden sind Basis-Dienstleister, die den
höherwertigen Services für ihre Realisierung dienen. Bei der verbotenen
```

```
Methodenliebe dagegen sind die Abhängigkeiten auf Basis der direkten Verwendung
so gewählt, dass zwei Methoden jeweils nur zusammen betrachtbar sind, d.h. die
Inhalte der einen Methode benötigen die Inhalte der anderen Methode und
umgekehrt. Dies hat zur Folge, dass beide Methoden jeweils nur monolithisch
zusammen bearbeitet werden können.

Implementierung:

*/

SELECT DISTINCT
        funcOne.name as FunctionOne,
        typeOne.fullname as TypeOne,
        funcTwo.name as FunctionTwo,
        typeTwo.fullname as TypeTwo
FROM
        TAccesses as accessesOne inner join
        TAccesses as accessesTwo on (accessesOne.targetid = accessesTwo.functionid) inner join
        TFunctions as funcOne on (funcOne.id = accessesOne.functionid) inner join
        TFunctions as funcTwo on (funcTwo.id = accessesTwo.functionid) inner join
        TTypes as typeOne on (funcOne.classid = typeOne.id) inner join
        TTypes as typeTwo on (funcTwo.classid = typeTwo.id)
WHERE

        /* avoid duplicate entries */
        funcOne.id < funcTwo.id and

        accessesTwo.targetid = accessesOne.functionid
;
```

Lösungsstrategie

Die Behebung besteht i.d.R. aus dem Aufteilen der Methoden. Da der Kontrollfluss bereits so aufgebaut ist, dass die Methode sich anders verhält, wenn Sie von der aufgerufenen Methode selbst aufgerufen wird (da es sonst eine Endlosschleife wäre) besteht die Behebung daraus, die Methode in einen Client und Serverteil aufzusplitten und die Aufrufe entsprechend anzupassen.

2.1.51 Verbotene Paketliebe

Definition

Zwischen zwei Paketen besteht eine zyklische Abhängigkeit, da die Inhalte der Pakete sich jeweils gegenseitig für das Compilieren benötigen.

Beschreibung

Die verbotene Paketliebe ist der grobgranularste Zyklus in der Sammlung der verbotenen Lieben. Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten, die jeweils separat verstehbar, testbar und wartbar sind. Dies Grundprinzip der Verfeinerung gilt insbesondere für Pakete. Abhängigkeiten zu anderen Teilen, die zwangsläufig entstehen, um ein Gesamtsystem aufbauen zu können, werden i.d.R. in Form von Schichten aufgebaut, d.h. es gibt reine Dienstleister (z.B. I/O-Funktionalität), die von der nächst höherwertigen Schicht verwendet werden, die ihrerseits wieder Funktionalität einer noch höheren Schicht anbietet. Bei der verbotenen Paketliebe dagegen sind die Abhängigkeiten, die auf Basis der Abhängigkeiten der Inhalte der Pakete ermittelt werden, so gewählt, dass zwei Pakete jeweils nur zusammen betrachtbar sind, d.h. die Inhalte des einen Pakets benötigen die Inhalte des anderen Pakets und umgekehrt. Dies hat zur Folge, dass beide Pakete jeweils nur monolithisch zusammen bearbeitet werden können.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Architekturebene

Informale Erkennung

- Ermittlung aller derjenigen Paketpaare mit direkter zyklischer Abhängigkeit. Als Abhängigkeiten werden sowohl Includes und Imports als auch direkte Verwendungen von Artefakten aus dem anderen Paket aufgefaßt.
- Jedes so ermittelte Paketpaar stellt eine Problemmusterinstanz dar.

Erkennung mit SISSy

```
/* $Id: VerbotenePaketliebe.sql,v 1.1 2006/01/18 15:25:32 seng Exp $ */
/* FZI Forschungszentrum Informatik */

/*
Definition:
Zwischen zwei Paketen besteht eine zyklische Abhängigkeit, da die Inhalte der
Pakete sich jeweils gegenseitig für das Compilieren benötigen.

Beschreibung:
Die verbotene Paketliebe ist der grobgranularste Zyklus in der Sammlung der
verbotenen Lieben. Bei einer geschickten "Divide et impera"-Strategie entstehen
modulare Einheiten, die jeweils separat verstehbar, testbar und wartbar sind.
Dies Grundprinzip der Verfeinerung gilt insbesondere für Pakete. Abhängigkeiten
zu anderen Teilen, die zwangsläufig entstehen, um ein Gesamtsystem aufbauen zu
können, werden i.d.R. in Form von Schichten aufgebaut, d.h. es gibt reine
Dienstleister (z.B. I/O-Funktionalität), die auf der nächst höherwertigen
Schicht verwendet werden, die ihrerseits wieder Funktionalität einer noch
höheren Schicht anbietet. Bei der verbotenen Paketliebe dagegen sind die
Abhängigkeiten, die auf Basis der Abhängigkeiten der Inhalte der Pakete
ermittelt werden, so gewählt, dass zwei Pakete jeweils nur zusammen betrachtbar
sind, d.h. die Inhalte des einen Pakets benötigen die Inhalte des anderen Pakets
und umgekehrt. Dies hat zur Folge, dass beide Pakete jeweils nur monolithisch
zusammen bearbeitet werden können.

Implementierung:
look for pairs of packages
- access each other's classes
- or their classes inherit from classes of the other
- duplicate pairs must be avoided

Note: optimized for performance (AT)

(OS) Pakete aus Bibliotheken werden nicht mehr gezählt

*/

CREATE  TABLE VKnowsRel AS
        SELECT DISTINCT
                res.classId
                , res.className
                , res.knownClassId
                , res.knownClassName
        FROM (
                /* Inheritance relations */
                SELECT DISTINCT
                        cls.id AS ClassId
                        , cls.fullName AS ClassName
                        , knowncls.id AS KnownClassId
                        , knowncls.fullName AS KnownClassName
                FROM
                        TConstants con
                        , TAccesses acc
                        , TTypes cls
                        , TTypes knowncls
                WHERE
                        con.name = 'TYPEACCESS_INHERITANCE'
                        AND con.value = acc.kindOfAccess
                        AND acc.sourceId = cls.id
                        AND acc.targetId = knowncls.id

                UNION

                /* Exception declarations AND casts */
                SELECT DISTINCT
                        cls.id AS ClassId
                        , cls.fullName AS ClassName
                        , knowncls.id AS KnownClassId
                        , knowncls.fullName AS KnownClassName
                FROM
```

```
                TConstants con
                , TAccesses acc
                , TTypes cls
                , TMembers m
                , TTypes knowncls
        WHERE
                (con.name = 'TYPEACCESS_THROW' OR
                con.name = 'TYPEACCESS_CAST')
                AND con.value = acc.kindOfAccess
                AND acc.functionId = m.id
                AND m.classId = cls.id
                AND acc.targetId = knowncls.id

        UNION

        /* Attributes AND properties */
        SELECT DISTINCT
                cls.id AS ClassId
                , cls.fullName AS ClassName
                , knowncls.id AS KnownClassId
                , knowncls.fullName AS KnownClassName
        FROM
                TTypes AS cls
                , TMembers AS m
                , TVariables AS var
                , TAccesses AS acc
                , TTypes knowncls
        WHERE
                cls.id = m.classId
                AND m.id = var.id
                AND var.typeDeclarationId = acc.id
                AND acc.targetId = knowncls.id

        UNION

        /* Formal parameters, catch parameters AND local variables */
        SELECT DISTINCT
                cls.id AS ClassId
                , cls.fullName AS ClassName
                , knowncls.id AS KnownClassId
                , knowncls.fullName AS KnownClassName
        FROM
                TTypes AS cls
                , TMembers AS m
                , TVariables AS var
                , TAccesses AS acc
                , TTypes knowncls
        WHERE
                cls.id = m.classId
                AND m.id = var.functionId
                AND var.typeDeclarationId = acc.id
                AND acc.targetId = knowncls.id

        UNION

        /* Return type */
        SELECT DISTINCT
                cls.id AS ClassId
                , cls.fullName AS ClassName
                , knowncls.id AS KnownClassId
                , knowncls.fullName AS KnownClassName
        FROM
                TTypes AS cls
                , TMembers AS m
                , TFunctions AS func
                , TAccesses AS acc
                , TTypes knowncls
        WHERE
                cls.id = m.classId
                AND m.id = func.id
                AND func.returnTypeDeclarationId = acc.id
                AND acc.targetId = knowncls.id

        UNION

        /* Accesses to members */
        SELECT DISTINCT
                cls.id AS ClassId
                , cls.fullName AS ClassName
                , knowncls.id AS KnownClassId
```

```
                        , knowncls.fullName AS KnownClassName
                FROM
                        TTypes AS cls
                        , TMembers AS sm
                        , TAccesses acc
                        , TMembers AS dm
                        , TTypes knowncls
                WHERE
                        cls.id = sm.classId
                        AND acc.functionId = sm.id
                        AND acc.targetId = dm.id
                        AND dm.classId = knowncls.id
        ) AS res
        /* no library classes */
/*      JOIN TSourceEntities AS s ON (res.classId = s.id AND s.assemblyFileId = -1) */
                , TTypes AS types
                , TConstants AS c
        WHERE
                res.classId != res.knownClassId
                AND res.knownClassId = types.id
                AND types.kindOfType = c.value
                AND (
                        c.name like '%CLASS'
                        OR c.name like '%INTERFACE'
                )
;


CREATE Table VPackageKnowsRel AS
        SELECT DISTINCT
                classA.packageid AS package1_ID
                , classC.packageid AS package2_ID
        FROM
                TTypes classA join
                TSourceEntities classASource on (classA.id = classASource.id) join
                TFiles classAFile on (classAFile.id = classASource.sourcefileid) join
                VKnowsRel vKnowsRel on (vknowsRel.classid = classA.id) join
                TTypes classB on (vknowsRel.knownclassid = classB.id) join
                TSourceEntities classBSource on (classB.id = classBSource.id) join
                TFiles classBFile on (classBFile.id = classBSource.sourcefileid) join
                TTypes classC on (classB.packageid = classC.packageid) join
                TSourceEntities classCSource on (classC.id = classCSource.id) join
                TFiles classCFile on (classCFile.id = classCSource.sourcefileid) join
                VKnowsRel vKnowsRel2 on (vknowsRel2.classId = classC.id) join
                TTypes classD on (vknowsRel2.knownClassId = classD.id) join
                TSourceEntities classDSource on (classD.id = classDSource.id) join
                TFiles classDFile on (classDFile.id = classDSource.sourcefileid)
        WHERE
                classA.packageid = classD.packageID
                AND classB.packageid != classA.packageID
                AND classC.packageID != classD.packageID

UNION

SELECT
        iType.packageid,
        imp.targetid
FROM
        TImports imp join
        TConstants iCon on (imp.kindoftarget = iCon.value) join
        TFiles iFiles on (imp.fileid = iFiles.id) join
        TSourceEntities iSource on (iSource.sourcefileid = iFiles.id) join
        TTypes iType on (iType.id = iSource.id)
WHERE
        iCon.name like 'PACK_%'

UNION

SELECT
        iType.packageid,
        importedType.packageid
FROM
        TImports imp join
        TConstants iCon on (imp.kindoftarget = iCon.value) join
        TFiles iFiles on (imp.fileid = iFiles.id) join
        TSourceEntities iSource on (iSource.sourcefileid = iFiles.id) join
        TTypes iType on (iType.id = iSource.id) join
        TTypes importedType on (imp.targetid = importedType.id)
WHERE
        iCon.name like 'TYPE_%'
```

```
;


SELECT DISTINCT
       packages1.fullname AS package1
       , packages2.fullname AS package2
FROM
       VPackageKnowsRel rel_1
              JOIN TPackages packages1 ON (packages1.id = rel_1.package1_ID)
       , VPackageKnowsRel rel_2
              JOIN TPackages packages2 ON (packages2.id = rel_2.package1_ID)
WHERE
       rel_1.package1_ID = rel_2.package2_ID
       AND rel_1.package2_ID = rel_2.package1_ID
       /* avoid duplicates */
       AND rel_1.package1_ID < rel_2.package1_ID
;

DROP TABLE VPackageKnowsRel;
DROP TABLE VKnowsRel;
```

Lösungsstrategie

Dieses Problem entsteht als direkte Folge einer verbotenen Klassen- oder Methodenliebe. Die Behebung erfolgt durch die Anwendung der entsprechenden Behandlungsstrategie.

2.1.52 Versteckte Konstantheit

Definition

Ein Klassen-Attribut (static) wird bis auf die initiale Wertesetzung bei der Deklaration (mittels Attribut-Initialisierer) nur lesend verwendet, ohne dass es als Konstante (final bzw. const) deklariert ist.

Beschreibung

Das Verstehen von Designs und Code ist umso effizienter, je mehr implizit vorhandenes Wissen der Programmierer mittels sprachspezifischer Konstrukte expliziert hat. Diese Grundannahme bezieht sich insbesondere auf die Unterscheidung von Attributen und Konstanten: Während Attribute ihre Inhalte/Werte zur Laufzeit ändern können und damit Zustände festhalten, dienen Konstanten lediglich dazu, im Vorfeld der Programmausführung feststehende Constraints abzubilden. Diese semantische Unterscheidung sollte programmiersprachenspezifisch so expliziert werden, daß Konstanten auch als Konstanten deklariert werden und nicht als Attribute.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene, Codeebene.

Informale Erkennung

- In einem ersten Schritten werden alle diejenigen Klassen-Attribute gesucht, die bis auf die Deklaration nur lesend verwendet werden.
- Anschließend werden aus dieser Ergebnismenge diejenigen Klassen-Attribute selektiert, die nicht als Konstante deklariert sind.
- Jedes so ermittelte Attribut stellt eine Problemmusterinstanz dar.

Erkennung mit SISSy

```
/* $Id: VersteckteKonstantheit.sql,v 1.1 2006/01/18 15:25:31 seng Exp $ */
/* FZI Forschungszentrum Informatik */
```

```
/*
Definition:
Ein Klassen-Attribut (static) wird bis auf die initiale Wertesetzung bei der
Deklaration (mittels Attribut-Initialisierer) nur lesend verwendet, ohne dass es
als Konstante (final bzw. const) deklariert ist.

Beschreibung:
Das Verstehen von Designs und Code ist umso effizienter, je mehr implizit
vorhandenes Wissen der Programmierer mittels sprachspezifischer Konstrukte
expliziert hat. Diese Grundannahme bezieht sich insbesondere auf die
Unterscheidung von Attributen und Konstanten: Während Attribute die Inhalte zur
Laufzeit ändern können und damit Zustände festhalten, dienen Konstanten
lediglich dazu, im Vorfeld der Programmausführung feststehende Constraints
abzubilden. Diese semantische Unterscheidung sollte programmiersprachenspezifisch
so expliziert werden, daß Konstanten auch als Konstanten deklariert werden und
nicht als Attribute.

Implementierung:
select all static attributes of system defined classes that are not final
except
select all static attributes of system defined classes that are not final and
there is a write access to them
*/

select distinct
       C.id as Class_ID,
       C.fullName as Class_Name,
       a.id as Attribute_ID,
       a.name as Attribute_Name
from
       TTypes as C,
       TMembers as a,
       TSourceEntities as seC,
       TConstants as constField
where

       /* C is a system defined class */
       seC.id = C.id and
       seC.sourceFileId >= 0 and

       a.kindOfMember = constField.value and
       constField.name = 'VAR_FIELD' and
       a.classId = C.id and
       a.isFinal = 0 and
       a.isStatic = 1

/* take out those that have a write access  pointing to them */
except

select distinct
       C.id as Class_ID,
       C.fullName as Class_Name,
       a.id as Attribute_ID,
       a.name as Attribute_Name
from
       TTypes as C,
       TMembers as a,
       TSourceEntities as seC,
       TConstants as constField,
       TAccesses as acc,
       TConstants as constVarWrite,
       TFunctions as accBelongsToFunction,
       TConstants as funcType
where

       /* C is a system defined class */
       seC.id = C.id and
       seC.sourceFileId >= 0 and

       a.kindOfMember = constField.value and
       constField.name = 'VAR_FIELD' and
       a.classId = C.id and
       a.isFinal = 0 and
       a.isStatic = 1 and

       /* there's a write access to a */
       acc.kindOfAccess = constVarWrite.value and
       constVarWrite.name = 'VARACCESS_WRITE' and
       acc.targetId = a.id AND
```

```
        /* No access in initializer */
        acc.functionid = accBelongsToFunction.id AND
        funcType.value = accBelongsToFunction.kindoffunction AND
        funcType.name <> 'FUNC_INITIALIZER'
;
```

Lösungsstrategie

Das Problem wird behoben, indem die entsprechenden Attribute als Konstanten deklariert werden. Hierzu muss die Schreibweise entsprechend der verwendeten Programmiersprache angepasst werden und ggfs. noch der Ort der Konstante systemkompatibel geändert werden.

2.2 Problemmuster mit manueller Erkennung

2.2.1 Allgemeine Attribute

Definition

Attribute werden sehr allgemein gehalten, ohne dass diese Verallgemeinerung genutzt wird.

Beschreibung

Der Wunsch ein Attribut innerhalb einer Klasse möglichst universell nutzbar zu machen führt oft dazu, dass dieses Attribut sehr allgemein gehalten wird. Im Extremfall wird als Attributtyp nur Objekt (Java) oder void (C++) verwendet.
Wird diese Verallgemeinerung dann nicht konsequent genutzt, sondern erfolgt stattdessen in manchen Fällen vor der Verwendung des Attributs ein Cast auf einen spezielleren Datentypen, dann ist die Verallgemeinerung unnötig und sollte durch den speziellen Datentypen ersetzt werden.
Beispiel (Java): Eine Klasse verwaltet eine sortierte Liste von Personennamen (namelist). Der Datentyp der Liste ist jedoch nicht als SortedSet (Interface für sortierte Mengen) sondern allgemein als Collection (Interface für Sammlungen, Oberklasse von SortedSet) angegeben. Initialisiert wird namelist jedoch als TreeSet (Implementierung/Unterklasse von SortedSet). Bis zu hier spricht noch nichts gegen die Deklaration von namelist als Collection.
Nun wird jedoch in der Implementierung das Interface von SortedSet verwendet, wozu ein Cast notwendig ist. Jetzt wird in der Implementierung explizit die Verwendung eines speziellen Datentyps vorausgesetzt und der allgemeine Datentyp sollte ersetzt werden. Ein weiteres Festhalten am allgemeinen Typ kann sogar zu Laufzeitfehlern führen, etwa wenn namelist mit ArrayList (Implementierung/Unterklasse von Collection) initialisiert wird. Das führt zu keinem Compilerfehler, da ArrayList eine Unterklasse von Collection ist. Erfolgt jedoch zur Laufzeit ein Cast auf SortedSet wird dies mit einer ClassCastException abgebrochen.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Design

Informale Erkennung

- Typecast eines Attributs
- auch auf unterschiedliche Klassen innerhalb einer Vererbungshierarchie, z.B. cast von Collection auf SortedSet (Unterklasse von Collection) und TreeSet (Unterklasse von SortedSet)

Erkennung mit SISSy

```
/* $Id: AllgemeineParameter.sql,v 1.2 2006/01/24 15:26:16 seng Exp $ */
/*
This Query looks for places in source code, where formal parameters are
downcasted.

*/

SELECT DISTINCT
        variables.name as Parameter,
        types.name as ParameterType,
        types2.name as CastedToType,
        functions.name as OccuringInFunction,
        types3.name as OccuringInClass,
        files.pathname as OccuringInFile
FROM
        TAccesses accesses1 join
        TVariables variables on (variables.id = accesses1.targetid) join
        TConstants cons on (variables.kindofvariable = cons.value and cons.name =
'VAR_FORMALPARAM') join
        TAccesses accesses2 on (accesses1.sourceid = accesses2.sourceid and
                accesses2.position = accesses1.position - 1) join
        /* Get the type of the formal parameter */
        TAccesses accesses3 on (accesses3.sourceid = variables.id) join
        /* Only downcasts are considered to be problematic */
        TInheritances inheritances on (accesses3.targetid = inheritances.superid and
inheritances.classid = accesses2.targetid) join
        /* In order to get the type */
        TTypes types on (accesses3.targetid = types.id) join
        /* Get the type that parameter was casted to */
        TTypes types2 on (accesses2.targetid = types2.id) join
        TFunctions functions on (accesses1.functionid = functions.id) join
        TTypes types3 on (functions.classid = types3.id) join
        TSourceEntities sourceEntities on (types3.id = sourceEntities.id) join
        TFiles files on (sourceEntities.sourcefileid = files.id)
;
```

Lösungsstrategie

Die Behebung dieses Problems besteht in der Verwendung des speziellen Datentyps oder in der Beschränkung auf Interface des allgemeinen Typs, hierbei sind jedoch ggf. umfangreichere Modifikationen notwendig.

2.2.2 Altmodische Methode

Definition

Anstatt einer "neueren" Implementierung (in einer Unterklasse) wird die "veraltete"/überschriebene Implementierung einer Oberklasse verwendet.

Beschreibung

Eine Klasse UK überschreibt eine implementierte Methode einer Oberklasse OK. Eine Instanz der Klasse UK wird jetzt über ein Cast in die Oberklasse OK umgewandelt und es wird die überschriebenen Methode aufgerufen (z.B. ((OK) uk).meth()). Welche Implementierung kommt jetzt zur Anwendung? Je nach dem ob es sich bei der überschriebenen Methode um eine statische, virtuelle oder normale Methode handelt, ändert sich das Verhalten.
Daher gilt: Überschreibt eine Klasse UK eine implementierte Methode einer Oberklasse OK, so ist das Hochcasten der Klasse nicht erlaubt. Ebenso dürfen Unterklassen von UK allenfalls auf UK gecastet werden, nicht jedoch allgemeiner (im Sinne der Vererbunghierarchie) als UK.

Dieses Problemmuster wurde als manuell klassifiziert, da eine Erkennung der Problemstellen mit den verfügbaren Werkzeugen nicht möglich ist.

Programmiersprache

C++, Java

Betrachtungsebene

Designebene

Lösungsstrategie

Die Behebung dieses Problems geschieht entlang der folgenden Schritte:

- Ausfaktorisieren des Verhaltens der überschriebenen Methode in der Oberklasse
- Ersetzen des Casts durch Aufruf der ausfaktorisierten Methode

2.2.3 Breite Subsystemschnittstelle

Definition

Ein Subsystem besitzt keine expliziten Schnittstellen.

Beschreibung

Wenn fast alle Klassen eines Subsystems von außen benutzt werden können, entsteht eine hohe Abhängigkeit. Es empfiehlt sich die Schnittstelle des Subsystems dahingehend zu verändern, dass nur noch wenige Klassen von außen angesprochen werden müssen, um somit die Benutzung des Subsystems zu vereinfachen.

Dieses Problemmuster wurde als „manuell“ klassifiziert, da die Subsystemdefinition/-zuordnung nicht aus dem Quellcode extrahiert werden können.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Architekturebene

Lösungsstrategie

Wenn die Mehrheit der Klassen innerhalb eines Pakets immer nur gemeinsam verwendet werden können, beziehungsweise wenn komplexe Protokolle bei der Verwendung der Klassen/Schnittstellen in einem Subsystem zu beachten sind, ist es empfehlenswert das Entwurfsmuster facade zu verwenden um eine einheitliche Schnittstelle zum ganzen Subsystem zu ermöglichen. In den meisten Fällen wird die Anpassung der "Außenwelt" (der Nutzer des Subsystems) benötigt.

2.2.4 Datenklasse

Definition

Eine Datenklasse speichert Daten, stellt aber keine Funktionalität für deren Bearbeitung bereit.

Beschreibung

Eine Klasse, die nur aus Attributen und Zugriffsmethoden (get- und set-Methoden) für diese Attribute besteht. Die zu den Attributen gehörende Funktionalität ist in anderen Klassen implementiert, welche auf die Attribute über Zugriffsmethoden zugreifen und sie verändern.

Ein Beispiel für eine Datenklasse ist eine Datumsklasse, die lediglich drei Integer Attribute (Tag, Monat, Jahr) und entsprechende Zugriffsmethoden (z.B. setDay(int d), getDay()) beinhaltet. Weitere Funktionen wie eine Gültigkeitsprüfung des Datums sind nicht Bestandteil der Klasse.

Eine zuverlässige Erkennung von Datenklassen setzt voraus, dass Getter- und Setter-Methoden eindeutig identifiziert werden können. Da dies mit den verfügbaren Werkzeugen nicht möglich ist, wurde das Muster als manuell klassifiziert.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Lösungsstrategie

Methoden, die Ausschließlich auf den Daten der Datenklassen arbeiten sollten in diese verschoben werden. Alternativ könnten auch die Daten in die Klassen verschoben werden, in denen sie benutzt werden. Letzteres ist aber nur dann empfehlenswert, wenn sich die Methoden, die ein Attribut der Datenklassen verwenden alle in einer Klasse befinden.

2.2.5 Deplazierte Klasse

Definition

Eine Klasse befindet sich im falschen Paket.

Beschreibung

In einem Paket kommuniziert eine Klasse mit mehr Klassen aus einem anderen Paket als mit Klassen des eigenen Pakets. Diese Klasse sollte in das entsprechende andere Paket verschoben werden. Somit können automatisch Zyklen auf Paketebene vermieden werden.

Dieses Problemmuster wurde als manuell eingestuft, da die Definition zu einer hohen Anzahl von falsch positiven Meldungen führt.

Programmiersprache

C++, Java, C#, ~~Delphi~~

Betrachtungsebene

Designebene

Lösungsstrategie

Verschieben der Klasse in das Paket, in dem die Klassen liegen, die am meisten benutzt werden, vorausgesetzt dass die Semantik dieses Pakets mit derjenigen der Klasse besser übereinstimmt.

2.2.6 Deplazierte Methode

Definition

Eine Methode benutzt häufiger Attribute und/oder Methoden anderer Klassen als der eigenen Klasse.

Beschreibung

Im objektorientierten Design bilden Objekte Einheiten von Daten und diese Daten benutzende Funktionen. Aus diesem Prinzip resultiert eine enge Bindung (Kopplung) zwischen den Daten und Funktionen einer Klasse. Das Prinzip ist durchbrochen, wenn eine Methode häufiger Attribute und/oder Methoden anderer Klassen verwendet als die der eigenen Klasse. Dies kann ein Zeichen dafür sein, dass diese Methode besser zu einer der benutzten Klasse gehört als zu der in der sie sich befindet.

Dieses Problemmuster wurde als manuell eingestuft, da die Definition zu einer hohen Anzahl von falsch positiven Meldungen führt.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Lösungsstrategie

Verschiebung der Methode in die Klasse, welche die von ihr benutzten Attribute und Methoden enthält, vorausgesetzt, dass die Semantik der Klasse mit derjenige der Methode besser übereinstimmt.

2.2.7 Flaschenhalsklasse

Definition

Eine Flaschenhalsklasse referenziert sehr viele andere Klassen und wird auch selbst von sehr vielen Klassen referenziert.

Beschreibung

Klassen von denen viele andere Klassen abhängen (die von vielen Klassen benutzt werden) wirken sich negativ auf die Stabilität eines Systems aus. Begründet ist das darin, dass bereits minimale Änderungen in solchen Klassen eine Vielzahl von Änderungen/Anpassungen in den abhängenden Klassen bedingen können. Das System ist somit instabil. Es liegt daher im allgemeinen Interesse diese Klassen möglichst nicht zu verändern. Dem wirkt jedoch entgegen, dass diese Klassen von vielen anderen Klassen abhängen, die Notwendigkeit/Wahrscheinlichkeit einer Änderung dadurch also verstärkt/vergrößert wird. Aus diesem Grund sollten Flaschenhalsklassen möglichst vermieden bzw. aufgeteilt werden.

Dieses Problemmuster wurde als „manuell“ eingestuft, da z.B. das Entwurfsmuster „Facade“ die Entstehung von Flaschenhalsklassen fördert bzw. bedingt.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Lösungsstrategie

Die Strategie für die Behebung dieses Problems hängt von der Natur der Abhängigkeiten ab und kann sehr unterschiedlich sein. Im Allgemeinen ist es wünschenswert den Flaschenhals durch die Aufteilung der Klasse, bzw. durch die Verschiebung ihrer Funktionalität zu bekämpfen. Wenn die Abhängigkeit der Klassen von der Flaschenhalsklasse durch Vererbung entstanden ist (d.h. es existieren sehr viele Varianten von Spezialisierungen) kann man die Lösungsstrategie von 2.2.19 verwenden.

2.2.8 Flaschenhalssubsystem

Definition

Ein Subsystem hängt von vielen anderen Subsystemen ab und benutzt seinerseits viele andere Subsysteme.

Beschreibung

Ein Subsystem von dem viele andere Subsysteme abhängen, wirkt sich in Hinblick auf die Stabilität des gesamten Systems kritisch aus. Jede Änderung in dem Subsystem kann umfangreiche Änderungen in den abhängenden Subsystemen bedingen. Das Subsystem sollte daher möglichst stabil gehalten werden. Dem wirkt jedoch die Tatsache entgegen, dass das Subsystem selbst von vielen Subsystemen abhängt. Die Wahrscheinlichkeit, dass eine Änderung in einem anderen Subsystem eine Änderung im betrachteten Subsystem notwendig macht, ist umso größer je mehr Abhängigkeiten zu anderen Subsystem bestehen.

Dieses Problemmuster wurde als „manuell“ klassifiziert, da die Subsystemdefinition/-zuordnung nicht aus dem Quellcode extrahiert werden können.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Architekturebene

Lösungsstrategie

Die Situation kann verbessert werden indem man versuchen kann (wenn die Semantik es zulässt) das Flaschenhalssubsystem in mehrere Subsysteme aufzuteilen, oder Teile des Flaschenhalssubsystems in andere Subsysteme zu verschieben.

2.2.9 Implementierungsähnliche Klassen

Definition

Zwei Klassen sind implementierungsähnlich, wenn sie die gleiche Funktionalität bieten.

Beschreibung

Verschiedene Klassen haben Methoden mit verschiedenen Signaturen, aber derselben Funktionalität. Dieses gemeinsame Verhalten sollte ausfaktorisiert werden, um eventuelle Änderungen nur an einer Stelle durchführen zu müssen. Eventuell kann sogar die Signatur der Methoden vereinheitlicht werden und die Methode in eine gemeinsame Oberklasse extrahiert werden. Alternativ kann der betroffene Teil der Methoden per Delegation in eine Hilfsklasse ausgelagert werden.

Dieses Problemmuster wurde als „manuell“ klassifiziert, da die automatische Erkennung einer ähnlichen Implementierung nicht realisiert werden kann.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Lösungsstrategie

Falls nur die Schnittstellen der Methoden unterschiedlich sind, liegt es Nahe die Funktionalität über Delegation auszufaktorisieren. Eventuell muss hierzu zunächst eine geeignete neue Klasse erzeugt werden, möglicherweise existiert aber auch schon eine Klasse, in die die Methode semantisch passen würde. Ein Indiz hierfür könnte eine hohe Kopplung des betreffenden Quelltextes zu dieser Klasse sein. Die Schnittstelle anzupassen hat größere Auswirkungen auf die Nutzer der Klasse. Falls die Schnittstelle nur leicht abgeändert werden muss, könnte zusätzlich eine gemeinsame Schnittstelle geschaffen oder eine gemeinsame Oberklasse erzeugt werden.

2.2.10 Klassensippe

Definition

Eine Menge von Klassen stehen in wechselseitiger (in)direkter Beziehung.

Beschreibung

Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten, die jeweils separat verstehbar, testbar und wartbar sind. Dies gilt insbesondere für Klassen. Bei einer Klassensippe dagegen sind die Abhängigkeiten, die ausgehend von allen möglichen Abhängigkeitstypen ermittelt werden (z.B. Typzugriff, Methodenaufrufe, Attribut-/Konstantenverwendung usw.) so gerichtet, dass mehrere Klassen gegenseitig voneinander (in)direkt abhängig sind. Dies hat zur Folge, dass alle beteiligten Klassen jeweils nur monolithisch zusammen bearbeitet werden können. Besonders schwerwiegend sind Klassensippen, die Subsystemgrenzen überwinden.

Dieses Problemmuster wurde als „manuell“ klassifiziert, da Aufgrund der möglichen komplexen Vernetzung der beteiligten Klassen keine allgemein gültigen Lösungsstrategien angegeben werden können.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Lösungsstrategie

Im Allgemeinen, besteht die Möglichkeit Klassenabhängigkeiten mit Abhängigkeiten von Schnittstellen zu ersetzen, die die ursprüngliche Klasse implementiert.

2.2.11 Langer Kommentar

Definition

Ein Quelltextkommentar ist zu lang.

Beschreibung

An einer Stelle des Quelltextes steht ein langer Kommentar. Das kann daran liegen, dass die Bedeutung der Methode oder Klasse sonst nicht klar wird, weil sie zu komplex ist. Ebenfalls möglich ist, dass der lange Kommentar ein großes auskommentiertes Code-Fragment darstellt. Im ersten Fall ist der Kommentar ein Hinweis auf komplexen Code, der wahrscheinlich auch mit Komplexitätsmetriken gefunden werden kann. Im zweiten Fall kann es ein Hinweis auf lange Debugsessions sein, und bläht den Code auf jeden Fall auf.

Das Problemmuster wurde als „manuell“ eingestuft, da die Semantik von Kommentaren nicht automatisch geprüft werden kann.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Lösungsstrategie

Falls der Kommentar aus totem Code entsteht, sollte dieser gelöscht werden. Falls der Kommentar auf komplexen Code hinweist, kann die Lösungsstrategie des Problemmusters Labyrinthmethode verwendet werden.

2.2.12 Layoutfehler

Definition

Das Layout des Quelltextes ist inkonsequent und inkonsistent.

Beschreibung

Damit der Quellcode gut lesbar ist und wichtige Stellen schnell gefunden werden können, sollte er genauso sorgfältig geschrieben werden wie andere Texte auch. Zu den Layoutfehlern gehören insbesondere Einrückungen, die gemischt mit Tabs und Leerzeichen eingefügt wurden, fehlende, inkonsistente oder falsche Klammerung von Blöcken sowie die Zeilenlänge.

Folgende Dinge sollten konsequent und konsistent angewendet werden.

- Zeilenlänge
- Leerzeichen/Tabs
- Klammerung
- Positionierung
- Leerzeilen

Das Problemmuster wurde als „manuell“ eingestuft, da für eine Layoutüberprüfung das „Ziellayout“ konfiguriert werden muss. Eine automatische Prüfung ist deshalb nicht möglich bzw. auch bei Einsatz von Heuristiken fehlerbehaftet.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Lösungsstrategie

Die Behebung des Problems besteht in der Berücksichtigung der bestehenden Konventionen:

- Konsequente Klammerung auch von einzeiligen Blöcken
- Überarbeitung der Positionierung von Klammern und Einrückungen
- Kürzen der Zeilenlänge

2.2.13 Nachrichtenketten

Definition

Eine Methode greift auf Daten einer Bibliothek zu indem sie sich entlang der internen Struktur der Bibliothek bewegt.

Beschreibung

Für den Zugriff auf bestimmte Daten einer Bibliothek muss ein Client über Wissen von der internen Struktur der Bibliothek besitzen und verwenden. Durch entlanghangeln an dieser Struktur greift der Client auf die Daten zu. Die interne Struktur einer Bibliothek sollte nach Außen transparent sein (im Sinne von nicht sichtbar), um Änderungen an der Struktur zu ermöglichen. Der Zugriff auf die Daten sollte über Schnittstellen erfolgen.

Das Problemmuster wurde als „manuell“ eingestuft, da keine allgemein gültige und präzise Definition der Problemstruktur entwickelt werden konnte. Potentielle Problemstellen könnten zwar automatisch markiert werden, die Identifizierung als tatsächliches Problem wäre/ist jedoch nur manuell möglich.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Lösungsstrategie

Das eigentliche Problem hier ist das Verstoß gegen das bekannte Demeters Gesetz, das besagt dass eine Methode eines Objekts nur Methoden der folgenden Arten von Objekten Aufrufen darf:

- sich selbst
- Objekte, die als Argument übergeben wurden
- Objekte, die sie instanziiert
- durch Komposition direkt gekapselter Objekte

Die einfachste Lösung besteht darin, dass man neue Methoden in den Klassen der Bibliothek einführt, die höherwertigen Dienste für die Klienten des Bibliothek anbieten, so dass die einzigen Abhängigkeiten zur Schnittstelle der Bibliothek bestehen. Die Funktionalität der neuen Methoden wird aus den Klienten der Bibliothek herausfaktorisiert.

2.2.14 Paketsippe

Definition

Eine Menge von Paketen stehen in wechselseitiger (in)direkter Beziehung.

Beschreibung

Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten, die jeweils separat verstehbar, testbar und wartbar sind. Dies Grundprinzip der Verfeinerung gilt insbesondere für Pakete. Abhängigkeiten zu anderen Teilen, die zwangsläufig entstehen, um ein Gesamtsystem aufbauen zu können, werden i.d.R. in Form von Schichten aufgebaut, d.h. es gibt reine Dienstleister (z.B. I/O-Funktionalität), die auf der nächst höherwertigen Schicht verwendet werden, die ihrerseits wieder Funktionalität einer noch höheren Schicht anbietet. Bei der Paketsippe dagegen sind die Abhängigkeiten, die auf Basis der Abhängigkeiten der Inhalte der Pakete ermittelt werden, der Gestalt, dass eine Vielzahl von Paketen jeweils nur zusammen betrachtbar sind, d.h. die Inhalte eines beliebigen Pakets hängen (in)direkt von anderen Paketen ab. Dies hat zur Folge, dass alle Pakete der Sippe jeweils nur monolithisch zusammen bearbeitet werden können.
Besonders schwerwiegend sind solche Abhängigkeiten, die über Subsystemgrenzen hinausgehen.

Dieses Problemmuster wurde als „manuell“ klassifiziert, da Aufgrund der möglichen komplexen Vernetzung der beteiligten Pakete keine allgemein gültigen Lösungsstrategien angegeben werden können.

Programmiersprache

C++, Java, C#, ~~Delphi~~

Betrachtungsebene

Designebene

Lösungsstrategie

Eine Möglichkeit wäre es, Klassen in entsprechende Pakete zu verschieben, um Zyklen zwischen Packages und/oder Subsystemen zu beseitigen. Besteht ein Zyklus zwischen Klassen über Paketgrenzen hinaus, sollten diese aufgetrennt werden oder mittels des DIP (Dependency Inversion Principle) beseitigt werden. Im Schlimmsten Fall ist eine Neuüberdenkung der gesamten Architektur und der Subsystemaufteilung notwendig.

2.2.15 Peripherieabhängigkeit

Definition

Abhängigkeiten zur Peripherie wie z.B. SQL-Abfrage oder Aufrufe von DOS-Kommandos befinden sind quer über das System verstreut.

Beschreibung

Abhängigkeiten zur Peripherie können nicht generell vermieden werden. Falls sie unbedingt nötig sind, sollten sie an zentraler Stelle gekapselt, damit Änderungen bei Anpassung an eine veränderte Peripherie lokal erfolgen können. Zusätzlich erscheren weit verstreute Abhängigkeiten auch das Verständnis des Systems, da die Abhängigkeiten nicht mit einem Blick erfasst werden können. Überall im System vorkommende SQL-Abfragen sind wohl das typischste Beispiel für eine Peripherieabhängigkeit. Oft wird der Aufwand gescheut, eine explizite Datenbankzugriffsschicht zu erstellen.

Dieses Problemmuster wurde als „manuell“ klassifiziert, da die für die Erkennung verwendeten Suchstrings an die jeweilige Technologie und Umgebung angepasst werden müssen, z.B. aufgrund der Verwendung von unterschiedlichen SQL Dialekten.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Codeebene

Lösungsstrategie

Das Problem kann behoben werden, indem eine Variablisierung stattfindet, d.h. dass über Variablen mit der Peripherie kommuniziert wird. Bestenfalls sollten jedoch wohl definierte Schnittstellen (Wrapper) implementiert werden, die die Kommunikation mit der Peripherie übernehmen und entsprechende Aufrufen dieser Schnittstelle angepasst werden. Eine solche Schnittstelle ist somit auch maximal austauschbar, wenn sich in der Peripherie Änderungen ergeben.

2.2.16 Subsysteme ohne Schnittstellen

Definition

Ein Subsystem enthält keine ausgezeichneten Schnittstellen, z.B. in Form von abstrakten Klassen oder Interfaces.

Beschreibung

Größere Subsysteme sollten ihre Schnittstelle möglichst vollständig in Form von Interfaces, nicht durch Klassen (in Java), bereitstellen. Sämtliche für Klienten bestimmte Code-Bestandteile sollten auf Interface-Typen beruhen. Dieses Vorgehen hat zumindest 2 wesentliche Vorteile:

- Klienten hängen nicht unmittelbar von einer Implementierung ab. Dies erhöht die Flexibilität insgesamt. Implementierungen können leicht variiert werden (z.B. für Logging, Proxies, Leistungsvariationen).
- Interfaces können sauber mehrfach beerbt werden. Dies erleichtert die Modellierung.

Dieses Problemmuster wurde als „manuell“ klassifiziert, da die Subsystemdefinition/-zuordnung nicht aus dem Quellcode extrahiert werden können.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Architekturebene

Lösungsstrategie

Zur Behebung des Problems müssen abstrakte Schnittstellen für Subsysteme definiert werden. Hierzu können Fabrikmuster zur Implementierungen der Schnittstellen verwendet werden.

2.2.17 Subsystemsippe

Definition

Eine Menge von Subsystemen steht in wechselseitiger (in)direkter Beziehung.

Beschreibung

Bei einer geschickten "Divide et impera"-Strategie entstehen modulare Einheiten, die jeweils separat verstehbar, testbar und wartbar sind. Abhängigkeiten zu anderen Teilen, die zwangsläufig entstehen, um ein Gesamtsystem aufbauen zu können, werden i.d.R. in Form von Schichten aufgebaut, d.h. es gibt reine Dienstleister (z.B. I/O-Funktionalität), die auf der nächst höherwertigen Schicht verwendet werden, die ihrerseits wieder Funktionalität einer noch höheren Schicht anbietet.
Bei der Subsystemsippe dagegen sind die Abhängigkeiten, die auf Basis der Abhängigkeiten der Inhalte der Subsysteme ermittelt werden, der Gestalt, dass eine Vielzahl von Subsystemen jeweils nur zusammen betrachtbar sind, d.h. die Inhalte eines beliebigen Subsystems hängen (in)direkt von anderen Subsystemen ab. Dies hat zur Folge, dass alle Subsysteme der Sippe jeweils nur monolithisch zusammen bearbeitet werden können.

Dieses Problemmuster wurde als „manuell“ klassifiziert, da die Subsystemdefinition/-zuordnung nicht aus dem Quellcode extrahiert werden können.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Lösungsstrategie

Um Zyklen aufzubrechen gibt es zwei Standardverfahren [Fow00]. Im ersten Verfahren wird ein Subsystem so aufgeteilt, dass die zwei anderen Subsysteme von diesem abhängig sind. Eine zweite Möglichkeit besteht darin, die Schnittstelle eines Subsystems in ein Interface umzuwandeln, so dass eine Abhängigkeit in eine Schnittstellenabhängigkeit verwandelt wird.

2.2.18 Uneinheitliche Operationsgruppen

Definition

Methoden für technisch ähnliche Operationen verwenden kein einheitliches Namensschema.

Beschreibung

Viele Methoden von Klassen führen unabhängig vom Anwendungsgebiet technisch ähnliche Operationen aus. Namen für Methoden/Funktionen sollten so gebildet werden, dass daraus der Operationstyp leicht erkennbar wird.
Beispiele für Operationskategorien und jeweils übliche Namenspräfixe/-bestandteile sind:

Tabelle 1

Namen	Kategorie	Bedeutung
is, can, has	Testen	Testet den Zustands des Objekts
make, create, new, build	Erzeugung	Erzeugt neue Objekte
init, setup	Initialisierung	Zuweisen initialer Werte

Auch innerhalb einer Operationsgruppe können Namensschamta die weitere Orientierung erleichtern, z.B. für Suchoperationen:

Tabelle 2

Namen	Bedeutung
find	Eine Einheit suchen, Null zurückgeben falls nicht gefunden
select	mehrere Einheiten suchen und einen Iterator über die Ergebnismenge (auch leere Menge) zurückgeben
Any	irgendeine Einheit identifizieren
All	alle Einheiten identifizieren

Sofern mit den einzelnen Präfixen keine unterschiedlichen Bedeutungen verknüpft sind, sollte jeweils nur EINE Form für den gesamten Code verwendet werden.

Das Problemmuster wurde als „manuell“ eingestuft, da die Erkennungsstrategie an die jeweils verwendeten Namensschema angepasst werden müssen.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Lösungsstrategie

Das Problem wird behoben, indem die Namensgebung vereinheitlicht wird.

2.2.19 Zerbrechliche Basisklasse

Definition

Eine Oberklasse besitzt viele Unterklassen und verwendet selbst viele andere Klassen.

Beschreibung

Eine zerbrechliche Basisklasse bildet ebenfalls einen Flaschenhals für Änderungen. Dies ist insbesondere kritisch, wenn die Wartung der Ober- und Unterklasse in unterschiedlichen Teams erfolgt. Dies ist z.B. bei Frameworks wie der MFC von Microsoft der Fall. Die Folge davon war, dass viele Leute ihre Anwendungen überarbeiten mussten, sobald Microsoft eine neue Version der MFC herausgebracht hat.

Das Problemmuster wurde als „manuell“ eingestuft, da keine allgemein gültige und präzise Definition der Problemstruktur entwickelt werden konnte. Potentielle Problemstellen könnten zwar automatisch markiert werden, die Identifizierung als tatsächliches Problem wäre/ist jedoch nur manuell möglich.

Programmiersprache

C++, Java, C#, Delphi

Betrachtungsebene

Designebene

Lösungsstrategie

Um dieses Problem zu beheben muss man überprüfen, ob eine Aufteilung der Klasse in mehre unabhängige Konzepte oder in so genannte Mixins sinnvoll ist. Eine weitere Möglichkeit besteht darin, dass man die Vererbungshierarchie unterhalb der zerbrechlichen Klasse neu gestaltet (weniger breit macht).

Literatur

[BMMM98]	W. Brown, R. Malveau, H. McCormick und T. Mowbray: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, 1998
[Fowler00]	M. Fowler: Refactoring: improving the design of existing code. Addison-Wesley 2000
[LW94]	B. Liskov and J. Wing. Family Values: A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems, November 1994. http://citeseer.ist.psu.edu/liskov94family.html
[Riel96]	Arthur J. Riel: Object-Oriented Design Heuristics. Addison-Wesley 1996
[URLCloneAnalyzer]	http://cloneanalyzer.sourceforge.net/
[URLSISSy]	http://www.qbench.de