ISSN 2549-4805

PENGGUNAAN KELAS ABSTRAK DALAM RANGKA MENDUKUNG PENERAPAN PRINSIP SOLID PADA DIAGRAM KELAS HASIL EKSTRAKSI ONTOLOGI

Frencius, S.Kom., M.T.
Sekolah Tinggi Teknologi Bandung
Jl. Soekarno-Hatta No. 378 - Bandung
frenciusleonardus@yahoo.com

Abstrak

Dalam perancangan berorientasi objek, diagram kelas sering digunakan untuk merancang perangkat lunak. Ontologi dapat diekstraksi menjadi diagram kelas, tetapi rancangan perangkat lunak tidak cukup dengan hanya mengandalkan diagram kelas dari hasil ekstraksi ontologi, karena diagram kelas harus disesuaikan dengan kebutuhan dan diterapkan prinsip desain. Perancang perangkat lunak sebaiknya menerapkan prinsip SOLID pada diagram kelas tersebut. Dengan diterapkannya prinsip SOLID pada diagram kelas yang dihasilkan dari ekstraksi ontologi, perangkat lunak yang dihasilkan dapat beradaptasi dengan perubahan dan tidak sulit dipelihara di masa yang akan datang. Maksud dari penelitian ini adalah menganalisis penggunaan kelas abstrak dalam rangka mendukung penerapan prinsip SOLID, sehingga diagram kelas hasil dari ekstraksi ontologi dapat menjadi dasar dalam merancang perangkat lunak yang mendukung penerapan prinsip SOLID. Hasil yang didapatkan dari penelitian ini menunjukkan bahwa penggunaan kelas abstrak mendukung penerapan prinsip SOLID pada diagram kelas hasil ekstraksi ontologi.
Kata kunci :
Diagram kelas, ontologi, SOLID.

Abstract

In object oriented design, class diagram is usually used to design software. Ontologies can be extracted into class diagrams, however software design is not enough just relying on the class diagram from ontology extraction, because the class diagram should adapt to the software requirements and be applied the object oriented design principles. Software designer should apply the principles of SOLID to it. If the class diagram is applied the principles of SOLID, the software will not be difficult to be maintained in future and can adapt to change. The purpose of this study is to analyze the use of abstract class in order to support SOLID principles application, so that the class diagram extracted from ontology can be basis in designing software that supports application of SOLID principles. The result obtained from this study shows that the use of abstract class supports application of SOLID principles to the class diagram extracted from ontology.
Keywords :
class diagram, *ontology*, SOLID.

1. PENDAHULUAN

Perancangan perangkat lunak tidak hanya berguna untuk mempermudah dalam pembangunan perangkat lunak tetapi juga untuk menghasilkan kualitas perangkat lunak yang baik (Martin, 2000). Dalam perancangan berorientasi objek, diagram kelas adalah salah satu bentuk rancangan perangkat lunak. Diagram kelas yang dirancang dengan baik tentu akan menghasilkan perangkat lunak yang baik. Banyak cara yang dapat digunakan untuk membuat diagram kelas, ada yang memulai dari analisis dan desain berorientasi objek, ada pula yang

ISSN 2549-4805

menggunakan *tools* khusus untuk men-*generate* diagram kelas. Salah satu cara membuat diagram kelas adalah dengan ontologi. Ontologi pada domain tertentu dapat diekstraksi menjadi sebuah diagram kelas. Dapatnya ontologi diekstraksi menjadi diagram kelas, dikarenakan adanya kemiripan antara ontologi dan paradigma berorientasi objek (Siricharoen, 2008). Struktur kelas atau *knowledge* yang sudah tersedia pada ontologi dapat memudahkan proses identifikasi objek dalam pembuatan diagram kelas.

Diagram kelas yang dihasilkan ontologi murni dari hubungan setiap kelas pada ontologi. Rancangan perangkat lunak tidak cukup dengan hanya mengandalkan diagram kelas dari hasil ekstraksi ontologi. Diagram kelas tersebut sebaiknya diterapkan prinsip *object oriented design* SOLID dan disesuaikan dengan kebutuhan perangkat lunak. Dengan tidak diterapkannya prinsip SOLID, perangkat lunak akan sulit untuk diperlihara di masa yang akan datang (Hall, 2014). Perubahan pada perangkat lunak di masa yang akan datang dimungkinkan sekali akan terjadi, jika perangkat lunak tidak dapat beradaptasi dengan perubahan, maka perangkat lunak tersebut akan sulit dipelihara (Martin, 2014). SOLID adalah kependekan dari *Single responsibility principle, Open-closed principle, Liskov substitution principle, Interface segregation principle, and Dependency inversion principle*. Kelima prinsip ini merupakan prinsip dalam perancangan perangkat lunak pada paradigma berorientasi objek.

Maksud dari penelitian ini adalah menganalisis penggunaan kelas abstrak dalam rangka penerapan prinsip *object oriented design* SOLID pada diagram kelas yang dihasilkan dari ekstraksi ontologi. Sehingga rancangan perangkat lunak yang didapatkan dari hasil ekstraksi ontologi dapat beradaptasi dengan perubahan dan tidak sulit dipelihara di masa yang akan datang.

Pada penelitian ini, kelas yang didapatkan dari ontologi dijadikan kelas abstrak pada diagram kelas. Ontologi adalah deskripsi formal secara eksplisit dari *terms* di dalam suatu domain dan hubungannya dengan *terms* yang lain (Noy, 2001). Pada ontologi terdapat kelas, *data properties*, *object properties* yang mirip dengan kelas, atribut, dan *relationship* pada paradigma berorientasi objek, sehingga ontologi dapat diekstraksi menjadi kelas diagram. Representasi ontologi yang menjadi input adalah OWL (Web Ontology Language). Dari OWL, kelas, *data properties*, *object properties* akan dijadikan kelas, atribut, dan *relationship* pada diagram kelas. Diagram kelas akan diterapkan prinsip SOLID melalui penggunaan kelas abstrak. Setiap kelas yang didapatkan dari ontologi akan dijadikan kelas abstrak pada diagram kelas.

Hasil yang didapatkan dari penelitian ini menunjukkan bahwa penggunaan kelas abstrak mendukung penerapan prinsip SOLID pada diagram kelas hasil ekstraksi ontologi.

2. KAJIAN LITERATUR

2.1 Kelas Abstrak

Kelas abstrak adalah sebuah kelas yang tidak memiliki *instance.* Kelas abstrak dibuat agar subkelas dapat menambahkan sturuktur dan *behavior* kelas dengan cara meng-*implement* kelas (Booch, 1994). Kelas abstrak dan subkelasnya berhubungan melalui *inheritance* atau *is-a.* Subkelas yang meng-*extend* kelas abstrak dapat menentukan sendiri *behavior* kelas sesuai dengan kebutuhan kelas. Di dalam kelas abstrak, terdapat *abstract method* dan *non-abstract method*, setiap *abstract method* dapat di-*implement* oleh kelas yang meng-*extend* kelas abstrak.

2.2 SOLID

2.2.1 SINGLE RESPONSIBILITY PRINCIPLE (SRP)

Prinsip ini menjelaskanJ bahwa sebaiknya kelas hanya memiliki *singleresponsibility*terhadap kelas lain. Halini dimaksudkan agar perubahan atau modifikasi pada sebuah kelas, hanya disebabkan *request*yang diminta oleh sebuah kelas lain. Sebuah kelas yang memiliki *multiple responsibility* dapat menyebabkan *error*pada kelas lain yang menggunakan kelas tersebut.

ISSN 2549-4805

2.2.2 OPEN CLOSED PRINCIPLE (OCP)

Prinsip ini menjelaskan bahwa penambahan fungsi atau *behavior*pada suatu kelas sebaiknya tidak dengan memodifikasi kelas tetapi dengan cara meng-*extend*kelas. Penambahan *behavior*suatu kelas dengan cara memodifikasi kelas memungkinkan terjadinya *error*pada kelas tersebut dan kelas lain yang menggunakan kelas tersebut. Penggunaan *extension point* (Hall, 2014)yang tepat dapat mendukung prinsip ini.

2.2.3 LISKOV SUBSTITUTION PRINCIPLE(LSP)

Prinsip ini menjelaskan bahwa objek dari *subclass* dapat disubsitusi dengan objek dari *base class* dan *client class*yang memiliki *dependency*kepada *base class* dapat berjalan dengan baik ketika *client class* tersebut menggunakan objek dari *subclass*. LSP merupakan sebuah petunjuk untuk membuat hirarki *inheritance*yang tepat, dengan *client class* dapat menggunakan *base class* atau *subclass*tanpa harus memikirkan benar atau tidaknya hasil yang didapatkan (Hall,2014).

2.2.4 INTERFACE SEGREGATION PRINCIPLE (ISP)

Prinsip ini mengatakan bahwa sebuah kelas seharusnya tidak bergantung kepada *interface* atau abstraksiyang tidak ada hubungannya dengan kelas tersebut.Prinsip ini membahas gejala '*fat'interface*, yaitu terdapat kelas yang meng-*implement*sebuah *interface yang* tidak ada hubungannya dengan kelastersebut. Ketika sebuah kelas bergantung pada *interface*yang tidak digunakannya, maka kelas tersebut akan ikut berubah ketika *interface*berubah, dan kelas tersebut harus meng-*implementmethod*yang ada di kelas *interface*, sementara kelas tersebut tidak membutuhkannya. Dalam kondisi ini, sebaiknya kelas *interface*dipisahkan dari kelas yang sama sekali tidak menggunakan *interface*tersebut.

2.2.5 DEPENDENCY INVERSION PRINCIPLE (DIP)

Prinsip ini mengatakan bahwa *high level modul* sebaiknya tidak bergantung kepada *low level modul*, keduanya sebaiknya bergantung kepada abstraksi. *High level modul* adalah sebuah *modul*yang berisi inti, *policy*,dan model bisnis dari program, *sedangkan low level modul* tidak berisi inti dari program tetapi sebagai pendukung berjalannya program. Prinsip ini juga mengatakan bahwa sebaiknya abstraksi tidak bergantung pada kelas kongkrit, tetapi kelas kongkrit yang bergantung pada abstraksi.

2.3 Merging Ontologies for Object Oriented Software Engineering oleh Waralak V. Siricharoen.

Menurut Waralak (2008), ontologi dapat digunakan untuk membantu mengidentifikasi objek. Dengan menggunakan ontologi pada fase analisis dan desain, pencarian objek menjadi lebih sistematis. Waralak mengusulkan metodologi yang dapat digunakan untuk membuat diagram kelas dari ontologi. Proses awal di dalam metodologi tersebut adalah pencarian ontologi yang tersedia secara *online* dan memilih ontologi yang dibutuhkan. Setelah mendapatkan ontologi yang sesuai, ontologi akan diproses oleh sebuah program yang berfungsi untuk mengekstraksi kelas beserta komponennya. Kelas dari hasil ekstraksi ontologi akan ditampilkan dan *developer* dapat menentukan kelas mana yang akan dipilih menjadi bagian dari kelas diagram yang akan dibuat. Proses yang digunakan untuk mengkonversi ontologi menjadi kelas/objek adalah dengan *translate* kelas pada ontologi sebagai kelas, *objectproperty* sebagai *relationships*, dan *dataproperty* sebagai atribut.

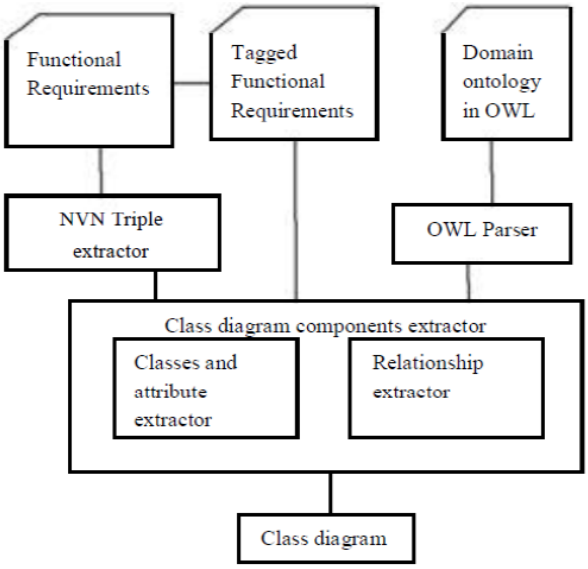
2.4 Domain Ontology Based Class Diagram Generation From Functional Requirements oleh Jyothilakshmi M. S. dan Philip Samuel.

Jyothilakshmi dan Philip(2012)membuat sebuah metode yang dapat digunakan untuk men-*generate*diagram kelasdari kebutuhan fungsional yang ditulis dalam bahasa natural. Metode ini menggunakan ontologi untuk

ISSN 2549-4805

mengidentifikasi kata kerja (*verbs*) dan kata benda (*nouns*) yang ada pada kebutuhan fungsional suatu perangkat lunak. Cara yang digunakan adalah dengan menghubungkan kata kerja dan kata benda tersebut ke ontologi. Ontologi akan membantu dalam mengidentifikasi termyang tidak disebutkan dalam kebutuhan fungsional.

Pada metode tersebut, hal pertama yang dilakukan adalah mem-*parsing*dokumen kebutuhan fungsional dengan menggunakan sebuah *parser*(SRS parser and tagger). Hasil yang didapatkan adalah kata benda dan kata kerja. Kata benda akan menjadi input untuk identifikasi kelas dan atribut, sedangkan kata kerja menjadi input untuk identifikasi operasi (*method*) dan *relationships*. Ontologi digunakan untuk mengidentifikasi kelas, atribut, operasi, dan *relationship*yang didapat dari proses *parsing* tersebut. Ontologi juga digunakan untuk menentukan pada kelas mana sebuah *relationship*berada, atau pada kelas mana sebuah atribut berada, atau pada kelas mana sebuah operasi berada. Secara detil, dokumen kebutuhan fungsional akan di-*parsing*oleh SRS Parser (OpenNLP), hasilnya adalah kata benda dan kata kerja. Kemudian sebuah ontologi pun di-*parsing*oleh OWL Parser (OWL API), hasilnya adalah *classes*, *ObjectProperty* (*relationship*), dan *DataProperty*(atribut). Kemudian ada yang disebut *classes and atributes extractor*, bagian ini akan mengekstraksi kelas dan atribut dari seluruh kata benda yang dihasilkan oleh SRS *parser*.Pada proses tersebut, ontologi digunakan. Jika terdapat kata benda yang sesuai dengan *DataProperty*pada OWL, maka kata benda tersebut adalah sebuah atribut. Hal yang sama dilakukan *relationships and operations extractor*, jika terdapat kata kerja yang sesuai dengan *ObjetProperty* pada OWL, maka kata kerja tersebut adalah *relationship*. Secara arsitektural, dapat dilihat pada gambar 1.

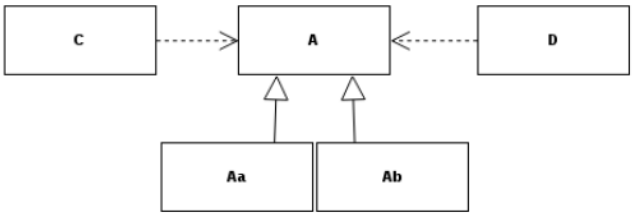


Gambar 1. Arsitektur ekstraksi diagram kelas oleh (Jyothilakshmi dan Philip, 2012)

3. ANALISIS

Pada bagianini dilakukan analisis penggunaan kelas abstrak dalam mendukung penerapan prinsip SOLID. Dari hasil analisis penggunaan kelas yang mendukung penerapan prinsip SOLID, disimpulkan bahwa penggunaan kelas abstrak mendukung setiap prinsip SOLID. Dengan menggunakan kelas abstrak pada diagram kelas, maka diagram kelas akan sesuai dengan setiap prinsip SOLID.

ISSN 2549-4805

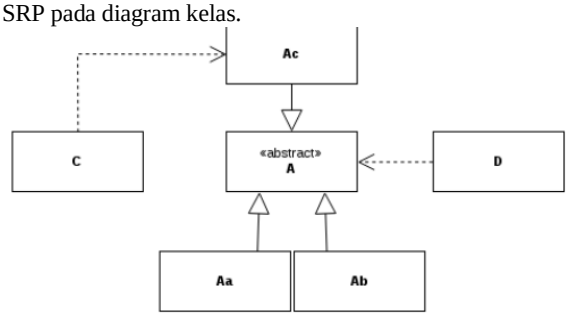


Gambar 2. Diagram kelas yang akan diterapkan prinsip

Diagram kelas pada Gambar 2 adalah sebuah diagram kelas sederhana yangakan diterapkan prinsip SOLID melalui penggunaan kelas abstrak. Hal ini dilakukan untuk menjelaskan lebih detil bagaimana penggunaan kelas abstrak dalam mendukung penerapan seluruh prinsip SOLID. Penerapan prinsip SOLID dibagi ke dalam beberapa subbab berikut ini.

3.1.1 PENERAPAN SRP

Penerapan SRP dengan cara menjadikan kelas menjadi kelas abstrak memudahkan dalam membagi *responsibility* suatu kelas. Berikut ini hasil penerapan SRP pada diagram kelas.



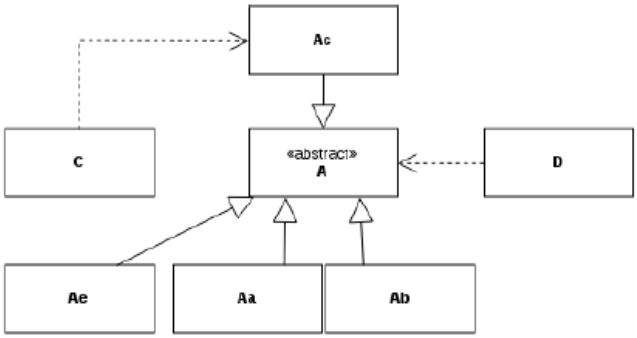
Gambar 3. Penerapan SRP dengan menggunakan kelas abstrak

Dengan menjadikan kelas A menjadi abstrak, maka *responsibility*kelas A dapat dibagi kepada kelas Ac melalui proses *extend*kelas. Pada dasarnya, hal ini tidak hanya bisa dilakukan oleh kelas abstrak, tetapi juga bisa dilakukan oleh kelas kongrit atau *interface*, tetapi jika *responsibility*kelas A dibagi dengan menggunakan kelas konkret, maka hal tersebut tidak mendukung penerapan ISP dan DIP. Hal ini disebabkan kelas konkret tidak mendukung penerapan ISP dan DIP. Sama halnya jika pembagian *responsibility*kelas A dilakukan dengan menggunakan *interface*, hal ini tidak akan mendukung penerapan LSP, karena *interface*tidak mendukung penerapan LSP. *Multiple responsibilities*pada setiap kelas sangat dimungkinkan sekali terjadi, maka dari itu, dengan menjadikan setiap kelas menjadi kelas abstrak, maka pembagian *responsibility*dapat dilakukan dengan mudah, dan hal ini tetap mendukung penerapan prinsip SOLID lainnya.

3.2 Penerapan OCP

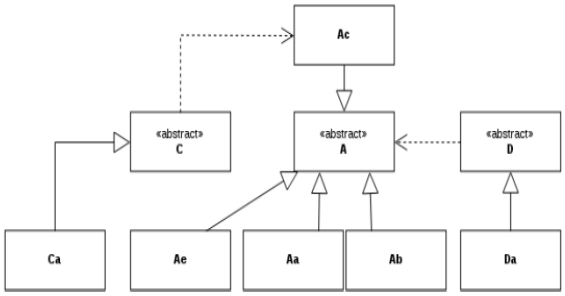
Penerapan OCP pada diagram kelas sama halnya dengan penerapan SRP, yaitu melalui penggunaankelas abstrak. Pada penerapan SRP, sebenarnya sudah diperlihatkan bagaimana penerapan OCP dilakukan, yaitu ketika kelas Ac melakukan *extend*terhadap kelas A. Dengan menjadikan kelas abstrak, maka penambahan *behavior*kelas dapat dengan mudah dilakukan dengan cara meng-*extend*kelas. Berikut ini dijelaskan penggunaan kelas abstrak dalam mendukung penerapan OCP.

ISSN 2549-4805



Gambar 4. Penerapan OCP dengan menggunakan kelas

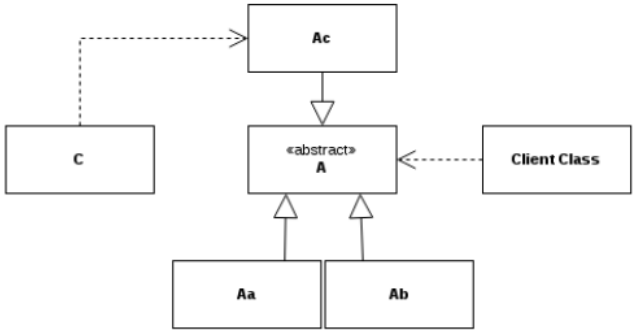
Kelas Ae adalah kelas baru yang ditambahkan melalui proses ekstensi kelas. Hal ini dapat dilakukan karena kelas A adalah kelas abstrak. Sama halnya dengan SRP, tidak hanya kelas abstrak yang mendukung penerapan OCP, tepai juga kelas konkret dan *interface*, tetapi jika menggunakan kelas konkret atau *interface*, maka hal tersebut tidak akan mendukung penerapan ISP dan OCP, atau LSP.Penambahan *behavior*suatu kelas dimungkinkan sekali akan terjadi ketika melakukan pengembangan terhadap diagram kelas, maka dari itu, dengan menjadikan setiap kelas menjadi kelas abstrak, maka penambahan *behavior*kelas dapat dilakukan dengan mudah dan hal ini tetap mendukung penerapan prinsip SOLID lainnya. Berikut ini dicontohkan penambahan *behavior* pada kelas C dan kelas D.



Gambar 5. Ca dan Da adalah kelas baru yang ditambahkan

3.3 Penerapan LSP

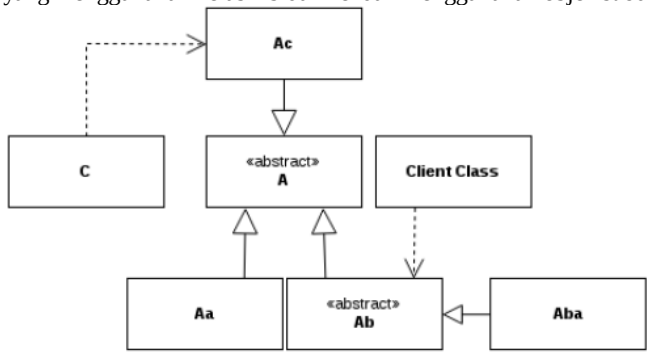
Penerapan LSP dilakukan dengan membuat *precondition*atau *postcondition*pada sebuah kelas agar objek dari *subclass*dapat disubtitusi dengan objek dari *base class*. Penggunaan kelas abstrak mendukung penerapan LSP. Dengan menggunakan kelas abstrak pada diagram kelas dan dengan hubungan *is-a* atau *inheritance*yang dimiliki *base class*dan *subclass*, maka substitusi objek dari *base class* dengan objekdari *subclass*dapat dengan mudah dilakukan.



Gambar 6. Penerapan LSP dengan kelas abstrak

ISSN 2549-4805

Pada Gambar 6, terdapat sebuah *client class* yang menggunakan kelas A. Dengan menggunakan kelas abstrak pada kelas A, maka substitusi objek antara kelas A dan *subclass*dari A (Aa dan Ab) dapat dilakukan. Pada dasarnya, hal ini juga dapat dilakukan dengan menggunakan kelas konkret. Dengan kelas konkret dan melalui hubungan *inheritance*, substitusi objek tentu bisa dilakukan. Tetapi, dengan menggunakan kelas konkret, tidak akan mendukung penerapan ISP dan DIP, karena ISP dan DIP tidak didukung melalui penggunaan kelas konkret. Penggunaan *interface*pun tidak dapat dilakukan karena penggunaan *interface*tidak mendukung penerapan LSP itu sendiri. Dengan menggunakan *interface*, tidak dapat dilakukan substitusi antara objek *subclass*dan objek *base class*, hal ini disebabkan objek dari *base class* tidak dapat dibuat jika *base class* adalah *interface*.
Substitusi objek pada setiap kelas dimungkinkan sekali terjadi, maka dengan menjadikan seluruh kelas menjadi kelas abstrak, maka penerapan LSP pada setiap kelas akan didukung, dan hal ini tidak mengganggu penerapan prinsip SOLID lainnya. Berikut ini terdapat sebuah *client class* yang menggunakan kelas Ab dan hendak menggunakan objek *subclass*dari Ab, yaitu Aba.



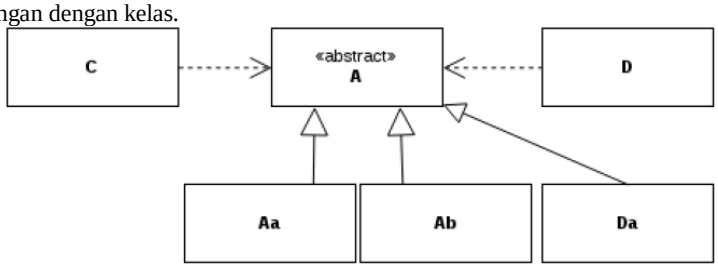
Gambar 7. Penggunaan kelas abstrak mendukung LSP

Dengan menjadikan seluruh kelas menjadi kelas abstrak, ketika ada sebuah *client class* yang ingin menggunakan objek *subclass*dari sebuah kelas, maka hal tersebut bisa dilakukan dan tidak akan mengganggu penerapan prinsip SOLID lainnya.

3.4 Penerapan ISP

Penerapan ISP pada diagram kelas adalah dengan memisahkan kelas dari *interface*atau kelas abstrak yang tidak ada hubungannya dengan kelas tersebut. Penggunaan kelas abstrak sangat mendukung penerapan ISP. Dengan adanya kelas abstrak, jika terdapat suatu kelas yang berada padasuatu kelas abstrak yang tidak digunakannya, maka kelas tersebut dapat dipindahkan pada kelas abstrak lain yang ada hubungannya dengan kelas tersebut atau dengan membuat sebuah kelas abstrak baru yang sesuai dengan kebutuhan kelas. Penggunaan interfacejuga mendungkung penerapan ISP, akan tetapi tidak mendukung penerapan LSP. Hal ini disebabkan karena *interface*tidak dapat digunakan untuk membuat objek, sedangkan penerapan LSP berhubungan dengan substitusi objek.

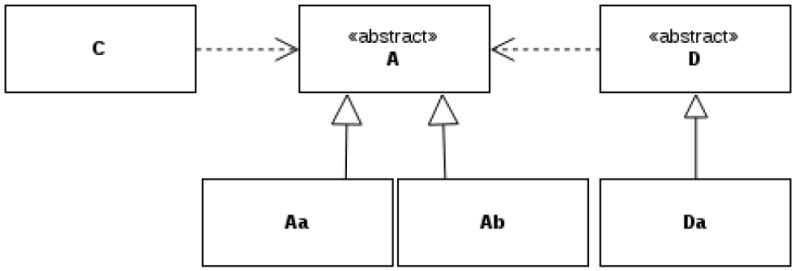
Adanya suatu kelas yang berada pada kelas abstrak yang tidak digunakannya sangat dimungkinkan terjadi, maka dengan menjadikan seluruh kelas menjadi kelas abstrak, kelas yang berada pada suatu kelas abstrak yang salah dapat langsung dipindahkan kepada kelas abstrak lain yang berhubungan dengan kelas. Berikut ini terdapat suatu kelas yang berada pada kelas abstrak yang tidak berhubungan dengan kelas.



Gambar 8. Kelas Da berada pada abstraksi yang tidak digunakannya

ISSN 2549-4805

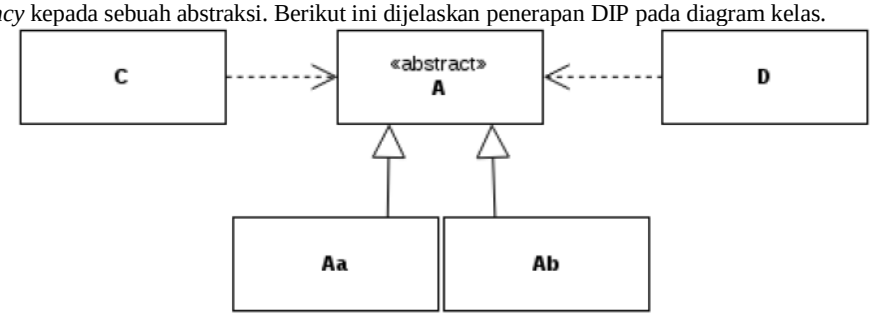
Kelas Da tidak berada pada abstraksi yang tepat, yaitu kelas A. Kelas Da adalah kelas D, maka dari itu kelas Da harus dipisahkan dari kelas A. Dengan menjadikan kelas D sebagai kelas abstrak, maka kelas Da dapat langsung dipindahkan kepada kelas D dan hal ini mendukung penerapan ISP. Jika kelas D adalah kelas konkret, maka kelas Da bukan berada pada sebuah abstraksi, melainkan pada kelas konkret. Berikutini pemisahan kelas Da kepada kelas D dengan menggunakan kelas abstrak.



Gambar 8. Dengan menjadikan kelas D abstrak, hal ini mendukung ISP

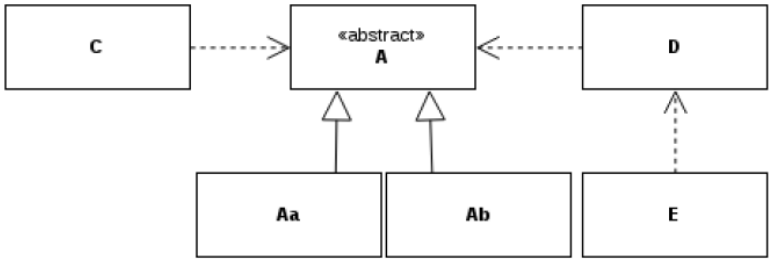
3.4.1 PENERAPAN DIP

Penerapan DIP dilakukan dengan mengubah *dependency high level modul*yang sebelumnya kepada kelas konkret menjadi kelas abstrak atau *interface*. Pada diagram kelas yang akan diterapkan prinsip SOLID (Gambar 9), kelas D memiliki *dependency*kepada kelas A. Jika diasumsikan bahwa kelas D adalah *high level modul* dan kelas A adalah *low level modul*, maka hal ini tidak sesuai dengan DIP, karena kelas D yang adalah *high level modul* bergantung kepada kelas A yang adalah *low level modul* atau kelas konkret. Maka hal yang perlu dilakukan adalah menjadikan kelas A menjadi kelas abstrak, sehingga kelas D memiliki *dependency* kepada sebuah abstraksi. Berikut ini dijelaskan penerapan DIP pada diagram kelas.



Gambar 9. Penerapan DIP dengan kelas abstrak

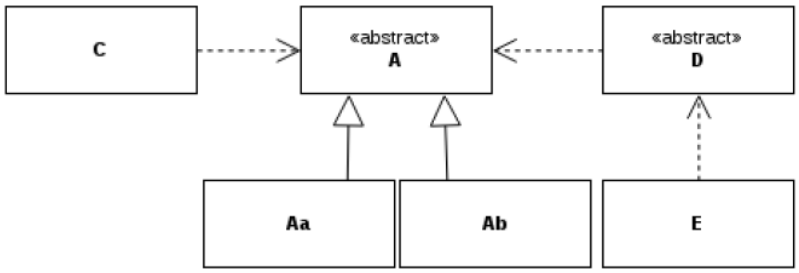
Selain dengan kelas abstrak, dijelaskan juga bahwa *interface*mendukung penerapan DIP, tetapi jika dengan menggunakan *interface*, maka akan mempengaruhi penepan LSP. *Interface*tidak mendukung penerapan LSP. Selain dari pada itu, dengan menjadikan seluruh kelas menjadi kelas abstrak, maka penerapan DIP pada setiap kelas dapat dengan mudah dilakukan. Penerapan DIP pada setiap kelas dimungkinkan sekali terjadi saat melakukan pengembangan terhadap diagram kelas. Berikut ini dijelaskan pengembangan diagram kelas dengan membuat sebuah kelas baru bernama kelas E. Kelas E adalah *high level modul* yang akan menggunakan kelas D.



Gambar 10. Kelas E memiliki *dependency* kepada kelas D

ISSN 2549-4805

Kelas E adalah sebuah kelas baru yang ditambahkan ketika melakukan pengembangan diagram kelas. Kelas E memiliki *dependency*terhadap kelas D. Jika kelas E adalah *high level modul*dan kelas E memiliki *dependency*kepada *low level modul* atau kelas konkret, maka hal ini tidak sesuai dengan DIP. Maka dari itu, dengan menjadikan seluruh kelas menjadi kelas abstrak, ketika ada suatu kelas yang akan ditambahkan, hal ini tetap mendukung penerapan DIP. Berikut ini ditampilkan diagram kelas yang sesuai dengan DIP.



Gambar 11. Penerapan DIP dengan kelas abstrak

4. KESIMPULAN DAN SARAN

Dari analisis dan percobaan penerapan yang dilakukan, dapat disimpulkan bahwa kelas abstrak sangat mendukung penerapan seluruh prinsip SOLID pada diagram kelas. Dengan menjadikan kelas menjadi kelas abstrak, maka tidak hanya satu prinsip saja yang didukung penerapannya, tetapi prinsip lainnya juga ikut didukung. Dijelaskan juga bahwa bahwa setiap kelas dimungkinkan sekali tidak sesuai dengan prinsip SOLID, maka dari itu, dengan menjadikan seluruh kelas menjadi kelas abstrak, maka dapat dipastikan setiap kelas mendukung penerapan prinsip SOLID. Berikut ini dirangkum alasan menjadikan seluruh kelas menjadi kelas abstrak.

1. Setiap kelas dimungkinkan memiliki *multiple responsibilities*, dengan menggunakan kelas abstrak, pembagian *responsibility*dengan mudah dilakukan dan tidak akan mengganggu penerapan prinsip SOLID lainnya.
2. Setiap kelas dimungkinkan untuk menambahkan *behavior*kelas, dengan menggunakan kelas abstrak, penambahan *behavior*dengan mudah dilakukan dan tidak akan mengganggu penerapan prinsip SOLID lainnya.
3. Setiap objek dari kelas dimungkinkan untuk disubstitusi dengan objek *subclass*-nya, dengan menggunakan kelas abstrak, substitusi objek dapat dilakukan dan tidak akan mengganggu penerapan prinsip SOLID lainnya.
4. Setiap kelas dimungkinan berada pada abstraksi yang tidak digunakan kelas, dengan menggunakan kelas abstrak, pemisahan kelas dari abstraksi yang tidak tepat dapat dengan mudah dilakukan dan tidak akan mengganggu penerapan prinsip SOLID lainnya.
5. Setiap kelas dimungkinkan memiliki *dependency*dengan *high level modul*, dengan menggunakan kelas abstrak, *high level modul*akan memiliki *dependency*terhadap abstraksi.

DATAR PUSTAKA

Martin, R. (2000): Design principles and design patterns. Object Mentor, diperoleh melalui situs internet: http://www.cogs.susx.ac.uk/users/ctf20/dphil_2005/Photos/Principles_and_Patterns.pdf\nhttp://scm0329.google.com/svnhistory/r78/trunk/book/Principles_and_Patterns.pdf. Diunduh pada tanggal 20 Oktober 2015.

Siricharoen, W. V. (2008): Merging Ontologies for Object Oriented Software Engineering. 2008 Fourth International Conference on Networked Computing and Advanced Information Management, 525–530.

Hall, G. M. (2014): Adaptive Code via C#: Agile coding with design patterns and SOLID principles.

Martin, R. C., dan Micah, M. (2006): *Agile Principles, Patterns, and Practices in C#*.

Noy, N. F., dan McGuinness, D. L. (2001): Ontology Development 101: A Guide to Creating Your First Ontology. Stanford Knowledge Systems Laboratory, 25.

Booch, G. (1994): *Object Oriented Analysis and Design with Applications, 2nd. Ed,*Addison-Wesley, California.

Jyothilakshmi, M. S., dan Samuel, P. (2012): Domain ontology based class diagram generation from functional requirements. *12th International Conference on Intelligent Systems Design and Applications (ISDA), 2012*, 6, 380–385.