ISBN: 979-756-061-6

CODE GENERATOR PYTHON DARI BENTUK XML METADATA INTERCHANGE (XMI) PADA UNIFIED MODELING LANGUAGE (UML) 2.0

I Made Murwantara, Pujianto Yugopuspito, Johan Julianto Roestam
Laboratorium Komputer Penelitian dan Pengembangan
Jurusan Teknik Informatika, Fakultas Ilmu Komputer, Universitas Pelita Harapan, Jakarta

Abstract

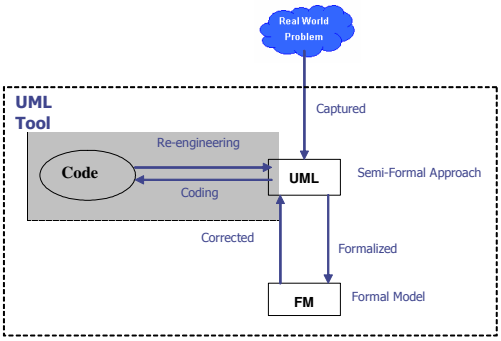
Unified Modeling Language (UML) code generator tools are available to parsing UML diagrams into source code. Currently, UML 2.0 tools that can parse into Python's source code are still in work planning phase. This research aims to create UML code generator that can parse XML Metadata Interchange (XMI) into Python's source code. XMI defines the design of program using UML 2.0 standard. Design process of this prototype uses UML standard version 2.0. On this research, algorithm for rules of parsing XMI into Python's source code was created.

***Keywords**: UML, Python, Parsing*

1. Pendahuluan

Object Management Group (OMG) membuat standarisasi perancangan piranti lunak. Implementasi dari standarisasi tersebut berupa terciptanya bahasa pemodelan yang dinamakan *Unified Modeling Language* (UML). UML merupakan bahasa pemodelan generasi ketiga yang digunakan untuk menentukan, menggambarkan, merancang, dan mendokumentasikan piranti lunak berorientasi objek (5). UML membangun model secara sistematik dan konsisten, baik dalam perancangan sistem secara sebagian maupun menyeluruh. UML tidak hanya menggambarkan ruang lingkup dari piranti lunak saja, tetapi menggambarkan pula *procedural* dan *business process* dari piranti lunak yang akan dibangun.

Ruang lingkup penulisan dapat dilihat pada gambar 1.



Gambar 1. UML *Interaction Tool*

Proses identifikasi masalah berawal dari proses pengumpulan kebutuhan untuk memperoleh deskripsi masalah secara keseluruhan. Dari masalah tersebut, UML akan membangun model dengan menggunakan pendekatan semi-formal. Pendekatan ini membatasi masalah yang ada dengan menggunakan diagram-diagram.

Diagram-diagram yang terbangun akan diperbaiki dalam *formal model*, berupa pembuktian secara matematika. Setelah diagram-diagram UML selesai diperbaiki, akan dipetakan ke dalam bentuk *source code* (2). Pada tahap pengkodean, terdapat dua proses pemetaan yaitu proses pemetaan diagram-diagram UML ke dalam bentuk XML *Metadata Interchange* (XMI) dan proses pemetaan XMI ke dalam bentuk kode bahasa pemrograman..

Object Management Group (OMG) telah mengemukakan standarisasi untuk mempercepat perkembangan teknologi piranti lunak berorientasi objek. Terdapat tiga standarisasi yang saling berhubungan, yaitu: *Meta Object Facility* (MOF), *Stream-based Model Interchange Format* (SMIF), dan *Unified Modeling Language* (UML).

Notasi grafik UML digunakan untuk menyatakan MOF *metamodel*. MOF mendefinisikan bahasa pemodelan yang digunakan dalam merancang suatu objek. SMIF menyediakan mekanisme pertukaran antar-elemen pemodelan berbasis *eXtensible Markup Language* (XML). Penamaan SMIF pun diganti dengan *XML Metadata Interchange* (XMI) seiring dengan berkembangnya versi dari SMIF. Pertukaran *metamodel* menggunakan XMI jauh lebih baik bila dibandingkan CORBA *interface*, karena *Object Request Broker* (ORB) tidak digunakan kembali. Dengan demikian, spesifikasi UML mendefinisikan UML *metamodel* sebagai MOF *metamodel*, sedangkan XMI diterapkan dalam pertukaran model UML.

Penggunaan diagram-diagram UML dalam membangun suatu model terbagi dalam tiga konsep dasar, yaitu: *model management diagram*, *structural diagram*, dan *behavioral diagram*. Ketiga konsep dasar ini dijabarkan pada table 1.

Dari tabel 1, terdapat peningkatan pada beberapa diagram pada UML 2.0 yang sebelumnya belum dideskripsikan secara jelas pada UML 1.3 (2). Diagram-diagram tersebut melengkapi bagian yang hilang dari *sequence diagram* dan memperjelas hubungan antar-diagram. Untuk lebih jelas perubahan diagram-diagram pada UML 2.0 dijabarkan pada tabel 2

ISBN: 979-756-061-6

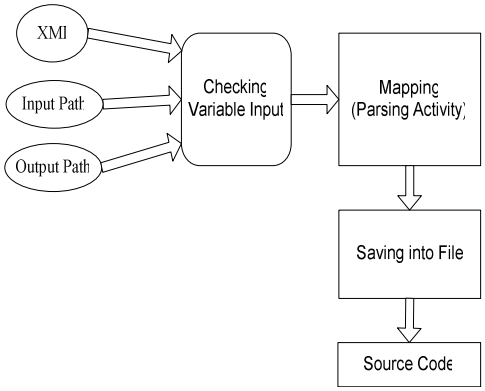
2. Mengapa Python

Bahasa pemrograman Python dipilih sebagai hasil akhir dari prototip yang dibangun tidak dikarenakan penelitian *code generator* ini masih dalam tahap *work planning* saja. Sebagai bahasa pemrograman berorientasi objek, Python memiliki beberapa keunggulan dibandingkan dengan bahasa pemrograman lainnya. Keunggulan dari Python secara umum dapat dijabarkan dari faktor-faktor berikut ini (3):

a. *Quality*; Python merupakan piranti lunak yang memakai metodologi *reusability* sehingga komponen-komponen pembangun piranti lunak mudah digunakan dan diatur kembali.
b. *Productivity*; penulisan program menggunakan Python lebih mudah, karena *interpreter* menangani *source code* secara terpisah pada *lower-level language*. Interpreter menangani tipe deklarasi variabel, manajemen *memory*, dan *procedure* yang digunakan dalam *source code*. Selain itu, penulisan program lebih mudah dimengerti, karena *syntax* Python mirip *executable pseudocode*.
c. *Portability*; program Python dapat dieksekusi di berbagai jenis komputer. Dengan demikian, proses eksekusi program dapat dilakukan tanpa mengubah *source code* program.
d. *Integration*; Python dirancang untuk dapat berinteraksi dengan aplikasi lain. Dengan demikian, program yang dibangun dengan bahasa pemrograman lain dapat dieksekusi dengan mudah pada *script* Python, menggunakan *function* bahasa pemrograman lain, mengakses COM *server*, dan lain-lain.

3. Pengembangan Tools

Prototip *Unified Modeling Language* (UML) *code generator* dirancang dengan menggunakan standarisasi UML 2.0. XML *Metadata Interchange* (XMI) merupakan elemen input terpenting dalam prototip ini. Dengan adanya XMI, prototip ini dapat menerima rancangan *project* dari aplikasi UML lain yang menyediakan fasilitas ekspor ke dalam bentuk XMI. Skenario jalannya prototip tersebut dideskripsikan pada gambar 2.



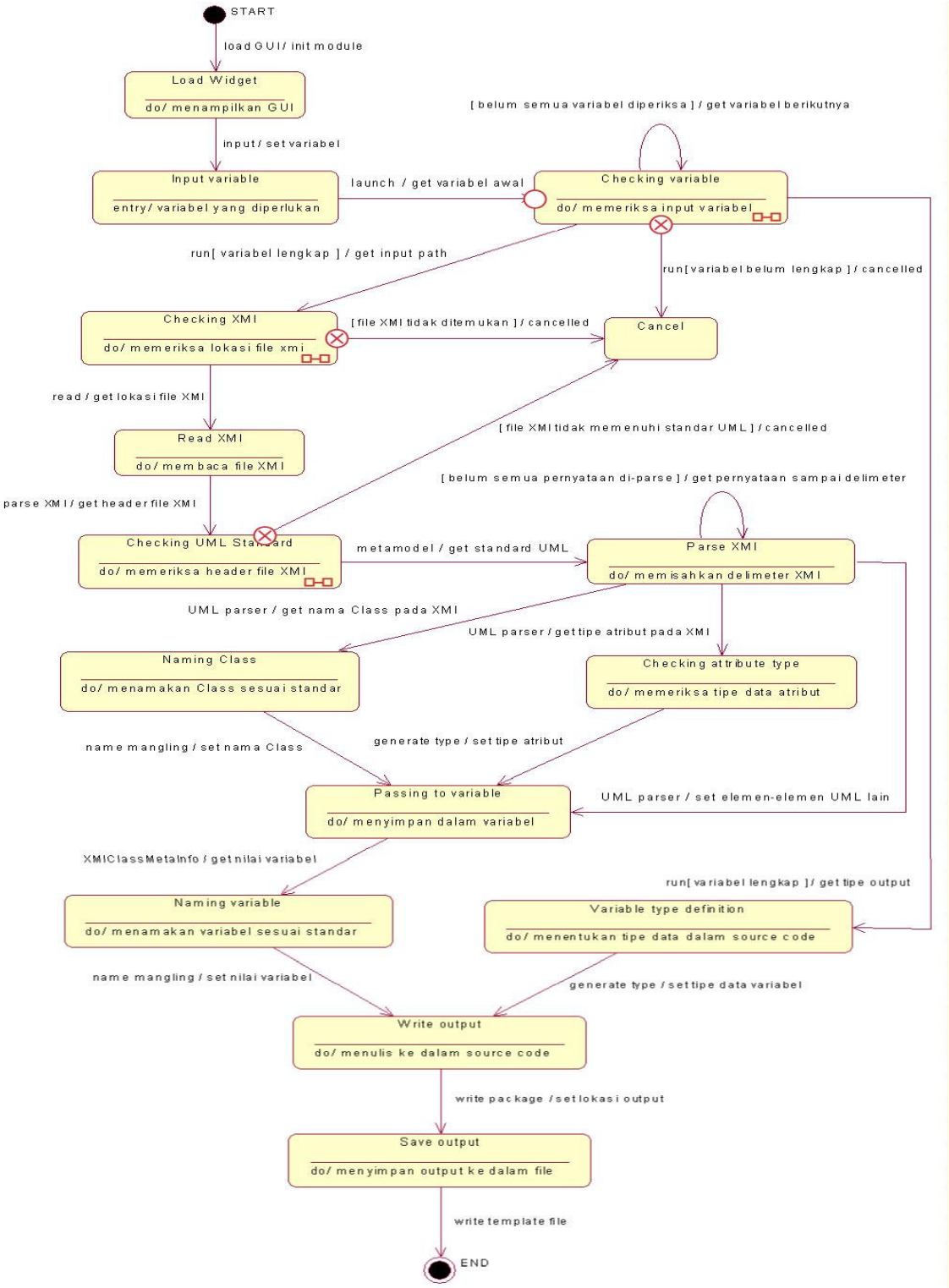
Gambar 2. Skenario Prototip UML *Code Generator*

Terdapat tiga elemen *input* yang diperlukan dalam prototip ini, yaitu: XMI, tipe *output*, dan lokasi *output*. Setelah *user* memasukkan ketiga elemen *input* ini, sistem akan memeriksa kelengkapannya. Bila elemen *input* tidak lengkap, *user* diberi kesempatan untuk memasukkan kembali elemen *input* yang belum terisi. Setelah semua elemen *input* lengkap, sistem akan melakukan pemetaan XMI dan hasilnya disimpan ke dalam variabel. Proses pemetaan XMI ini dilakukan dengan menggunakan *tree-based parser*. Metode ini digunakan karena mudah dan cepat dalam menganalisis data yang telah dipetakan (1). Nilai-nilai dalam variabel tersebut akan digunakan dalam penulisan *syntax* kode bahasa pemrograman Python. Proses penulisan *syntax* program tergantung dari tipe *output* yang dipilih oleh *user*. Setelah proses penulisan *syntax* program selesai, sistem akan menyimpan *source code* tersebut dalam bentuk *file*. Lokasi penyimpanan *file* tersebut ditentukan oleh *user*. Hasil akhir dari prototip ini berupa *source code* dalam bahasa pemrograman Python.

Skenario jalannya prototip di atas dituangkan dengan merancang diagram-diagram berbasis UML. Salah satu diagram pemodelan UML adalah *state machine diagram*. Perancangan prototip dengan menggunakan diagram tersebut dapat dilihat pada gambar 2.

Dari skenario prototip, elemen *input* yang sudah *user* masukkan diproses dalam empat tahap, yaitu:.

ISBN: 979-756-061-6



Gambar 4. *State Machine Diagram* pada *Code Generator*

ISBN: 979-756-061-6

a. Tahap Pemeriksaan Variabel

Sebelum masuk ke tahap ini, sistem akan memanggil *Graphical User Interface* (GUI) di saat program dieksekusi. Pada GUI, *user* perlu mengisi variabel-variabel yang diperlukan, seperti: *output module*, *input path*, dan *output* mengisi variabel-variabel yang diperlukan, seperti: *output module*, *input path*, dan *output path*. *Output module* adalah tipe *output* yang dipilih oleh *user*. *Input path* adalah lokasi XML *Metadata Interchange* (XMI) yang akan dipetakan dalam *code generator*. *Output path* adalah lokasi *output file* dari hasil *generate* yang akan disimpan ke dalam sistem operasi.

Setelah ketiga tipe variabel tersebut dimasukkan, proses pemeriksaan dilakukan dengan menjalankan operasi launch(). Operasi ini memeriksa isi dari ketiga tipe variabel. Bila salah satu dari nilai variabel tersebut tidak terisi, maka sistem akan berhenti dan memberi kesempatan untuk mengisi kembali. Bila seluruh variabel sudah terisi maka proses pemetaan XMI dijalankan dengan memanggil operasi read() pada *class* parseXMI. Deskripsi aliran proses dalam tahap ini dapat dilihat pada gambar 5.

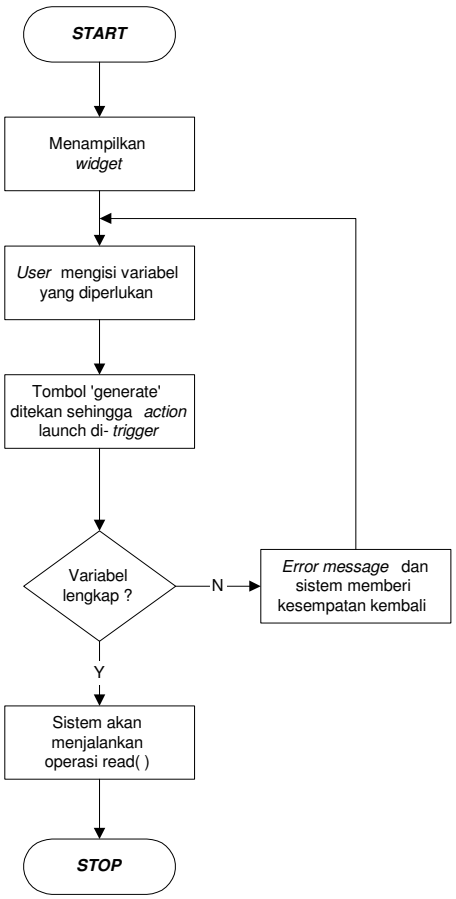
b. Tahap Pemetaan XMI

Pada tahap ini, sistem akan menjalankan operasi read(). Operasi ini digunakan untuk memeriksa lokasi dari XML *Metadata Interchange* (XMI) pada sistem operasi. Bila XMI telah ditemukan, operasi parseXMI() dijalankan. Bila XMI tidak memenuhi *syntax* pada standarisasi XMI, sistem akan berhenti. Operasi ini akan memanggil operasi-operasi lain. Pertama-tama sistem akan memanggil operasi UML13() dan operasi UML14() dalam *class meta*-model untuk melakukan inisialisasi dari standarisasi *Unified Modeling Language* (UML). Sistem hanya memilih satu dari kedua operasi tersebut sesuai dengan hasil pemetaan *header* XMI *file*.

Setelah memenuhi standarisasi UML, sistem mulai melakukan operasi parseXMIClass(). Operasi ini berisi operasi getUUID() untuk mengambil keunikan dari *base class* dan memanggil operasi-operasi seperti:

- Operasi parseXMIFeature(): operasi ini digunakan untuk memetakan atribut-atribut yang dimiliki oleh *class*.
- Operasi CapWord(): operasi ini digunakan untuk memperbaiki nama *class* yang sesuai dengan standarisasi. Penamaan *class* yang benar diawali dengan huruf kapital pada penulisannya.
- Operasi getAssociation(): operasi ini digunakan untuk mengambil *association* yang terdapat pada *base class*.
- Operasi parseXMIAssociation(): operasi ini digunakan untuk memetakan *association* beserta elemen-elemen yang ada.

Hasil pemetaan *class* akan disimpan ke dalam variabel yang berada pada *class* XMIClassMetaInfo, sedangkan hasil pemetaan *association* disimpan ke dalam variabel yang berada pada *class* XMIAssociationEnd. Kemudian *primary key* dan *foreign key* dimasukkan ke dalam *class* XMIClassMetaInfo. Dalam penentuan kedua *key* tersebut, sistem akan mengambil nama *class* yang telah dibubuhi dengan akhiran "_pk" untuk *primary key* atau "_fk" untuk *foreign key* bila pada diagram UML tidak ditentukan. Sistem pun mengubah penulisan tipe data dari atribut menjadi tipe *native* data dengan menjalankan operasi isNativeType(). Misalnya, tipe data char pada bahasa C menjadi tipe data character. Deskripsi aliran proses dalam tahap ini dapat dilihat pada gambar 6.

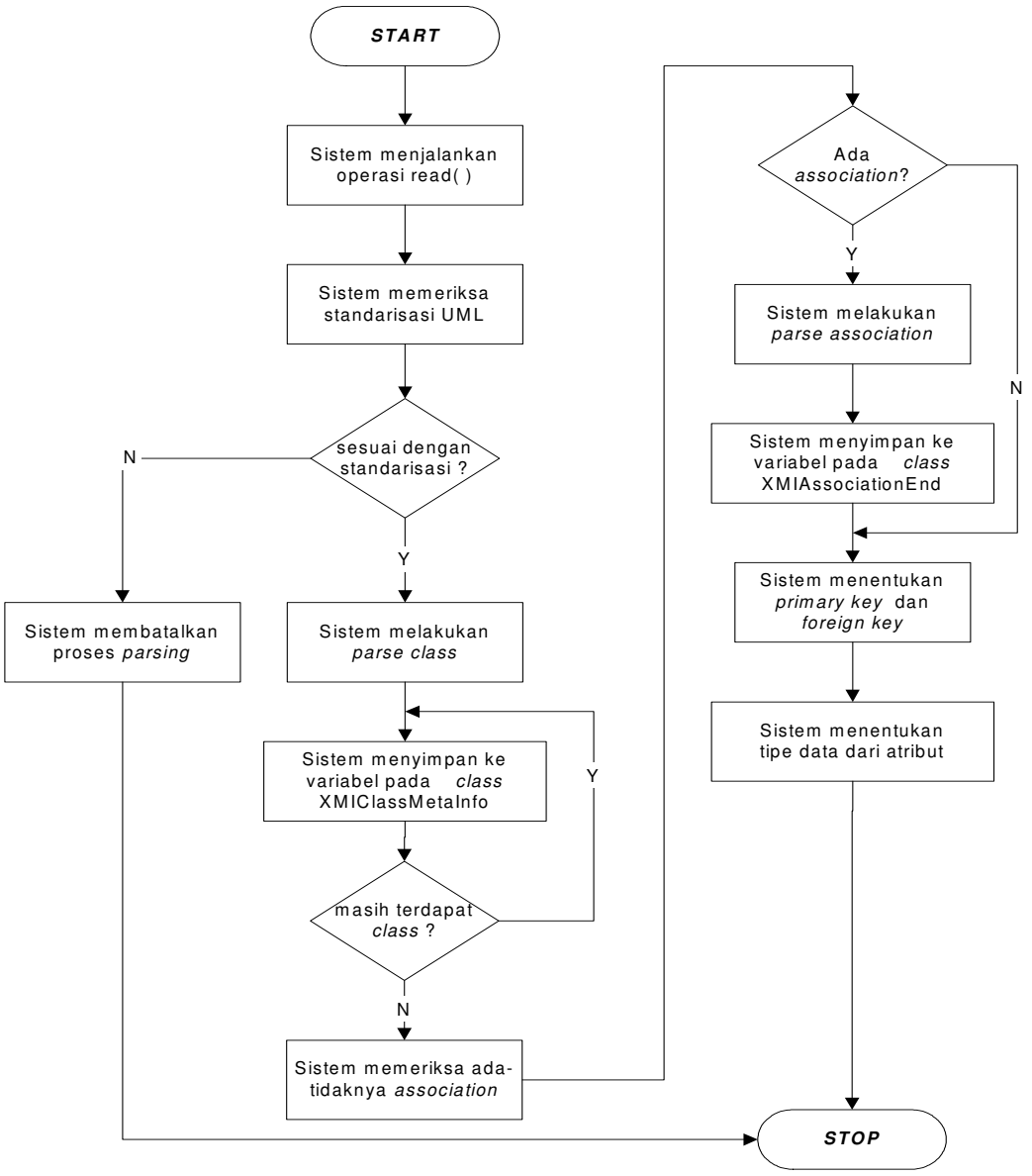


Gambar 5. *Flowchart Diagram* pada Tahap Pemeriksaan Variabel

c. Tahap penulisan ke bentuk bahasa pemrograman Python

Pada tahap ini sistem akan melakukan operasi write(). Operasi ini akan menyimpan *class* ke dalam bentuk simpul dengan memanggil operasi computeDependencyGraph() dan operasi forwardDeclarationSort(). *Class* akan disusun

 ISBN: 979-756-061-6



Gambar 6. Flowchart Diagram pada Tahap *Parse* XMI

berdasarkan *parent* dan *child*. *Parent* mewakili *base class*, sedangkan *child* mewakili *foreign class*.

Setelah simpul berbentuk *tree* tersusun, operasi writeClass() pun dilakukan. Operasi ini memanggil operasi getAttributes() dari *class* XMIClassMetaInfo. Setelah semua *attribute* yang diperlukan disimpan, operasi writeClass()memanggil operasi-operasi sebagai berikut:

- Operasi createHeader(): operasi ini digunakan untuk membuat *import file* yang diperlukan pada *source code*.
- Operasi createClassHeader(): operasi ini digunakan untuk membangun *template* dari *class* berisi nama *class* dan nama tabel untuk koneksi ke dalam *database*. Dalam proses perbaikan nama *class* dan tabel diperlukan operasi mangle() dalam *class* underscore_word dan *class* CapWord.
- Operasi createClassInit(): operasi ini digunakan untuk membangun fungsi __init__ yang berisi deklarasi variabel berupa atribut beserta tipe datanya dan *association* antar-*class*. Dalam inisialisasi tipe data dari setiap atribut diperlukan operasi getPythonTypeStr() dalam class GenerateType.
- Operasi createDocString(): operasi ini digunakan untuk menggeser empat spasi ke dalam yang menandakan isi dari *class*. Untuk

ISBN: 979-756-061-6

memproses tersebut diperlukan operasi addIndentToStrings().

Proses penulisan *class* ke dalam bentuk *source code* dijabarkan menjadi beberapa *class* utama, yaitu:

- Pada *class* createClassHeader, sistem menulis nama *class* dan tabel dari diagram *Unified Modeling Language* (UML). Variabel [baseClassString] memperoleh nilai dari variabel [classMetaInfo.getName(TRANSLATOR_NAME)]. Berikut ini potongan *source code* pada *class* createClassHeader:

```
class[classMetaInfo.getName(Translator_Name)] [baseClassString]:
    table_name                              =
classMetaInfo.getName(SQL_TRANSLATOR_NAME).
```

- Pada *class* createAccessors, sistem menulis fungsi set() dan get() untuk setiap atribut. Variabel [f.getGetterName(Translator_Name)] memperoleh nilai dari variabel field. Variabel [sqlName] memperoleh nilai dari variabel [f.getName(SQL_TRANSLATOR_NAME)]. Berikut ini potongan *source code* pada *class* createAccessors:

```
def [f.getGetterName(TRANSLATOR_NAME)]:
    return
   self.attributes["[sqlName]"].value
def
   [f.getSetterName(TRANSLATOR_NAME)](self, value):
    return
   self.attributes["[sqlName]"].setValue(value)
```

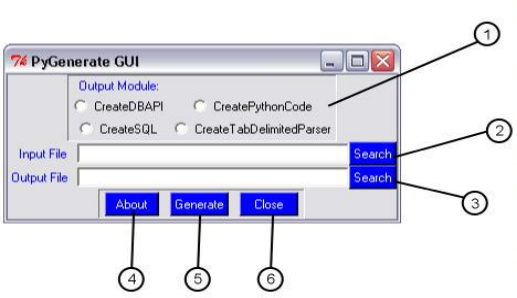
d. Tahap penyimpanan ke dalam bentuk file

Tahap ini sistem akan menyimpan *source code* ke dalam bentuk *file*. Sistem akan memanggil operasi-operasi berikut:

- Operasi writePackage(): operasi ini digunakan untuk menulis *package* beserta *support file* lainnya ke dalam *destination path*. *Package* yang disimpan berupa API.py, __init__.py, dan fkeyTypes.py.
- Operasi parsePackageInfo(): operasi ini digunakan untuk membangun *package* dari *class* tersebut berupa *import module*.
- Operasi writeTemplateFile(): operasi ini digunakan untuk menulis *template file* ke dalam *destination file*.

Prototip ini dilengkapi dengan *widget* sehingga *user* tidak perlu menghafal setiap perintah yang ada. Deskripsi *widget* dari prototip UML *code generator* dapat dilihat pada gambar 5. *Widget* ini dibangun pada *class* utama pyGenerator.py dengan menggunakan *library Tkinter*. prototip ini tidak hanya membangun aplikasi dalam Python saja. *User* dapat memilih *output module* yang diperlukan berupa:

a. CreateDBAPI: membangun koneksi akses ke dalam basis data berbasis Python.
b. CreatePythonCode: membangun aplikasi berbasis Python.
c. CreateSQL: membangun perintah-perintah *query* SQL berupa membangun tabel-tabel dalam basis data berbasis Python.
d. CreateTabDelimitedParser: membangun *parser* yang dapat mendefinisikan setiap *class* dalam diagram UML.



Gambar 7. Widget Python Code Generator

Keterangan:
1: *Output module*
2: *Input path*
3: *Output path*
4: *About programmer*
5: *Generate, parsing file XMI to Python's source code*
6: *Exit prototype code generator*

4. Kesimpulan

Pembuatan prototip dari *Unified Modeling Language* (UML) *code generator* melakukan pemetaan XML *Metadata Interchange* (XMI) ke dalam kode-kode bahasa pemrograman berbasis Python. Prototip ini terdiri dari lebih dari 18 *class*, yang tidak hanya menangani Python API saja. Prototip ini pun mampu menangani *Database* API, SQL *query*, dan *tab delimited parser*.

Daftar Pustaka

[1] Daconta, Michael C.; *Saganich, Al, XML Development with Java 2*, 1st edt, SAMS, USA, 2000, p. 40.
[2] Kotsis, G.; Bressan S.; et al, *The Sixth International Conference on Information Integration and Web-based Applications and Services (iiWAS2004)*, Oesterrelchische Computer Gessellchaft. 2004.
[3] Lutz, Mark, *Programming Python*, 2nd edt, O'Relly, 2001.
[4] Object Management Group, *XML Metadata Interchange (XMI) Specification*, www.omg.org/issues, Januari 2002.
[5] OMG UML Home, *Unified Modeling Language*, www.omg.org , Januari 2002.

ISBN: 979-756-061-6

LAMPIRAN

Tabel 1. Konsep Dasar UML

Major Area	*View*	*Diagrams*	*Main Concepts*
Structural	static view	class diagram	class, association, generalization, dependency, realization, interface.
	use case view	use case diagram	use case, actor, association, extend, include, use case generalization.
	implementation view	component diagram	component, interface, dependency, realization.
	deployment view	deployment diagram	node, component, dependency, location.
Dynamic / behavioral	state machine view	behavior state machine diagram	state, event, transition, action.
		protocol state machine diagram	input state, junction, composite state, terminate.
	activity view	activity diagram	state, activity, completion transition, fork join.
	interaction view	sequence diagram	interaction, object message, activation.
		communication diagram	collaboration, interaction, collaboration role, message.
		interaction overview diagram	flow control, information exchange.
		timing diagram	state, lifeline, event.
Model management	model view	class diagram	package, subsystem, model.

Table 2. UML 1.3 ke UML 2.0

UML 1.3	*Status*	*UML 2.0*
Sequence diagram	Remains	Sequence diagram
Collaboration diagram	Becomes	Communication diagram
Statechart diagram	Becomes	State machine diagram
Interaction diagram	New	Interaction overview diagram
	New	Timing diagram