

BAB I

PENDAHULUAN

1.1 Latar Belakang Masalah

Bahasa Jepang menggunakan berbagai jenis karakter untuk sistem menulisnya. Salah satu jenisnya adalah *kana*, yaitu karakter fonetis yang melambangkan bunyi silabel tanpa memberikan arti tertentu padanya. Contohnya adalah は yang melambangkan bunyi “ha” dan な yang melambangkan bunyi “na”. *Kana* dapat digabung untuk membentuk kata, sebagaimana pada alfabet latin. Sebagai contoh, はな dibaca “hana” dan bisa saja berarti “bunga”, “hidung”, maupun beberapa hal lain juga yang memiliki suara “hana”.

Ada dua jenis *kana* yaitu *hiragana* dan *katakana*. Sebagai contoh, *hiragana* dan *katakana* dari silabel “a” masing-masing adalah あ dan ア. Walaupun melambangkan suara yang sama, *hiragana* dan *katakana* digunakan untuk keperluan berbeda, sebagaimana pada huruf latin terdapat huruf besar dan huruf kecil yang masing-masing memiliki perannya sendiri.

Selain itu, huruf latin juga digunakan untuk kata-kata tertentu misalnya CD dan DVD. Untuk penulisan angka, sebetulnya Jepang memiliki sistem sendiri yang menggunakan karakter dari China. Walaupun begitu, yang lebih umum digunakan saat ini adalah bilangan Arab (0-9).

Terdapat satu lagi jenis karakter yang digunakan yaitu *kanji*. Seluruh karakter yang telah disebutkan tadi jumlahnya sangat sedikit dibandingkan dengan jumlah *kanji*. Karakter ini berasal dari China dan jumlahnya mencapai ribuan.

Suatu *kanji* memiliki sejumlah bunyi sekaligus arti yang intrinsik. Ini bisa dianalogikan seperti simbol “2” yang sudah memiliki bunyi sekaligus arti tertentu. Contoh *kanji* adalah 花 yang dibaca “hana” dan berarti bunga dan 鼻 yang juga dibaca “hana” namun berarti hidung.

Jumlah karakter yang mencapai ribuan tersebut menimbulkan masalah dalam proses *input* menggunakan *keyboard*, karena jumlah tombol *keyboard* terbatas. Untuk itulah terdapat program yang bernama *Input Method Editor* atau IME. IME pada umumnya adalah servis teks yang merupakan bagian dari sistem operasi. IME merupakan lapisan tambahan di antara pengguna yang memasukkan *input* dan aplikasi yang menunggu *input* dari pengguna. IME bahasa Jepang akan mengubah *input* dari keyboard menjadi teks Jepang yang diinginkan pemakai.

Sebagai contoh, dengan IME pengguna bisa menuliskan “hana” dan mengaktifkan tombol konversi (misalnya spasi). IME lalu akan mencoba memberikan suatu transliterasi misalnya “鼻” (hidung). Karena banyak kata bahasa Jepang yang memiliki bunyi “hana”, IME juga memberikan mekanisme bagi pengguna untuk memilih alternatif lainnya seperti “花” (bunga) dan “端” (ujung). Setelah pengguna memilih transliterasi yang diinginkannya, barulah hasil transliterasinya diserahkan ke program yang sedang aktif.

Di Jepang, komputer umum (misal komputer perpustakaan dan warnet) bisa dipastikan memiliki IME. Ini karena keberadaan IME adalah hal yang penting bagi aktivitas komputer orang Jepang. Di luar Jepang, kebutuhan menulis bahasa Jepang tidaklah umum, sehingga IME bahasa Jepang tidak diinstall di komputer umum. Hal ini menimbulkan kesulitan bagi segelintir orang yang benar-

benar membutuhkannya. Contohnya adalah orang Jepang yang sedang berkunjung ke luar negeri dan pelajar bahasa Jepang yang ingin berkomunikasi di Internet menggunakan bahasa Jepang.

Teknologi internet sudah berkembang sedemikian rupa sehingga aplikasi-aplikasi yang dulunya hanya ada di desktop kini sudah dapat dibuat sebagai aplikasi web. Contohnya adalah Google Docs dan Google Spreadsheet yang merupakan program pengolah kata dan program spreadsheet berbasis web. Dengan teknologi seperti DHTML dan AJAX, dimungkinkan pembuatan IME berbasis web yang membutuhkan antarmuka yang cukup kompleks. IME berbasis web dapat diakses dari mana saja dan tidak memerlukan instalasi yang rumit, sehingga dapat menjadi penyelamat saat IME desktop tidak tersedia.

1.2 Rumusan Masalah

Berdasarkan uraian latar belakang, bisa ditetapkan bahwa masalah dalam karya tulis ini adalah bagaimana membuat IME bahasa Jepang berbasis web dengan menggunakan teknologi web standar.

1.3 Batasan Masalah

Batasan masalah yang telah ditetapkan dalam tugas akhir ini adalah:

1. Yang dikembangkan adalah IME berbasis fonetis di mana pengguna memberikan transliterasi kata yang diinginkan menggunakan huruf latin dan sistem memberikan alternatif penulisannya dengan huruf Jepang.
2. Tujuan sistem adalah memungkinkan penulisan huruf Jepang bagi mereka

yang sudah memahaminya. Secara spesifik, pengguna tidak mungkin menggunakannya sebagai suatu kamus.

3. Yang dapat ditransliterasi adalah kata dasar dan berbagai infleksinya, dengan penambahan opsional partikel dan *gobi*. Untuk menulis suatu kalimat, pengguna dapat melakukan transliterasi per kata.
4. Kelengkapan kata dasar yang dikenali adalah selengkap kamus EDICT yang digunakan program ini.
5. Tata bahasa untuk infleksi didefinisikan di berkas teks sehingga walaupun belum lengkap dapat dengan mudah disempurnakan.
6. Pengujian sisi klien dilakukan pada Firefox 2, Opera 9, dan Internet Explorer 6. Kemampuan *browser* lain untuk menjalankannya tidak dijamin.
7. Font untuk bahasa Jepang diasumsikan telah terinstall di komputer pengguna. Tanpanya, karakter-karakter Jepang akan tampil sebagai kotak, tanda tanya, atau bentuk lain yang tidak diinginkan.
8. Teknologi yang digunakan di sisi klien adalah HTML, CSS, JavaScript, dan AJAX. Di sisi server digunakan C# (versi 2.0), ASP.NET (versi 2.0), dan SQLite (versi 3).

1.4 Tujuan Penelitian

Tujuan yang ingin dicapai dalam penelitian ini adalah membuat sebuah *Input Method Editor* bahasa Jepang sebagai aplikasi web. Aplikasi web ini dibangun menggunakan teknologi standard web sehingga klien dengan *browser*

modern dapat menggunakannya tanpa perlu menginstall program tambahan misalnya Java atau Flash.

1.5 Manfaat Penelitian

Dengan IME yang dikembangkan, pengguna komputer umum di luar Jepang dapat menulis teks bahasa Jepang dengan mengunjungi suatu situs. Karena menggunakan teknologi web standard, pengguna dapat langsung menggunakannya tanpa perlu menginstall program apapun.

1.6 Tinjauan Pustaka

Tae Kim's Japanese guide to Japanese grammar merupakan panduan yang sangat baik untuk tata bahasa Jepang. Panduan tersebut tersedia di internet dan dapat didownload oleh siapapun (Kim). Pembahasan mengenai tata bahasa Jepang yang sangat lengkap dengan format ensiklopedis bisa ditemui di *Japanese, A Comprehensive Grammar* (Kaiser dkk., 2001).

Terdapat beberapa IME yang berjalan sebagai bagian dari sistem operasi. Informasi mengenai Microsoft IME yaitu IME untuk Windows bisa didapat di situsnya (<http://www.microsoft.com>). Salah satu IME yang populer untuk desktop Linux adalah SCIM-anthy (<http://www.scim-im.org>). Taka Kudo dari NTT Communication Science Laboratories telah sebelumnya membuat IME berbasis web (Kudo) yang merupakan antarmuka terhadap IME mecab-skkserv yang berjalan di server. Dibandingkan dengan program yang dikembangkan penulis, aplikasi Taka Kudo mampu melakukan segmentasi kalimat walaupun batasnya

tidak bisa diubah pengguna jika batas yang diberikan tidak sesuai dengan yang diinginkan. Di lain pihak, aplikasi yang dikembangkan penulis menyimpan respons AJAX pada suatu *cache* sedangkan aplikasi Taka Kudo tidak.

Panduan ke HTML dan CSS bisa didapatkan di situs *w3schools* (<http://w3schools.com>). Pengenalan pada JavaScript bisa didapatkan di *Professional JavaScript for Web Developers* (Zakas, 2005). Topik-topik lanjut JavaScript, termasuk pengenalan dan penggunaan AJAX, bisa ditemui di *Pro JavaScript Techniques* (Resig, 2006).

Pengalan singkat ke C# dalam bahasa Indonesia bisa ditemui di *Bahasa C# untuk Pemrograman Berorientasi Objek* (Tien, 2001). Buku C# yang lebih komprehensif adalah *C#: A Beginner's Guide* (Schildt, 2001). Salah satu buku pemrograman ASP.NET adalah *ASP.NET 2.0 Unleashed* (Walther, 2006).

Buku yang membahas algoritma secara umum, termasuk DFS yang digunakan pada Gama IME, adalah *Introduction to Algorithms* (Cormen dkk., 2001).

Dokumentasi mengenai kamus EDICT bisa diperoleh di situsnya (<http://www.csse.monash.edu.au>). Informasi tentang DBMS SQLite bisa didapat di situsnya (<http://sqlite.org>).

1.7 Metodologi Penelitian

Untuk menyelesaikan tugas akhir ini, metodologi yang digunakan adalah sebagai berikut:

1. Studi literatur mengenai sistem menulis dan tata bahasa Jepang.

Pengetahuan tentang perubahan-perubahan kata yang mungkin dalam bahasa Jepang diperlukan sebelum bisa merancang sistem yang dapat melakukan transliterasinya. Studi literatur juga dilakukan tentang teknologi-teknologi yang diperlukan untuk membuat programnya, misalnya JavaScript dan format kamus EDICT.

2. Perancangan sistem yang terdiri dari 2 bagian yaitu bagian klien dan server. Perancangan algoritma dan struktur data untuk transliterasi disesuaikan dengan pengetahuan mengenai perubahan kata pada bahasa Jepang yang telah dipelajari sebelumnya dan informasi yang disediakan oleh kamus EDICT. Sisi klien dirancang agar menghasilkan antarmuka yang semirip mungkin dengan IME desktop.
3. Mengimplementasikan sistem dengan HTML, CSS, JavaScript, dan AJAX di sisi klien dan C#, ASP.NET, dan SQLite di sisi server.
4. Penulisan tugas akhir, di mana sistem yang telah dibuat akan dibahas.

1.8 Sistematika Penulisan

Sistematika penulisan yang digunakan untuk laporan ini adalah:

BAB I: PENDAHULUAN

Bab ini menjelaskan tentang latar belakang masalah, rumusan masalah, batasan masalah, tujuan penelitian, manfaat penelitian, tinjauan pustaka yang digunakan sebagai bahan referensi dalam penulisan tugas akhir, metode penelitian yang dipakai, dan sistematika penulisan tugas akhir.

BAB II: LANDASAN TEORI

Bab ini memuat teori-teori yang berhubungan dengan penelitian, mulai dari sistem tulis bahasa Jepang, gambaran mengenai perubahan kata yang mungkin pada tata bahasanya, format kamus EDICT, dan tinjauan umum tentang teknologi-teknologi yang digunakan seperti HTML, CSS, dan JavaScript.

BAB III: ANALISIS, PERANCANGAN, DAN IMPLEMENTASI

Bab ini memuat mulai dari analisis sampai dengan implementasi tiap bagian sistem. Pada sisi server, yang dibahas adalah sistem transliterasi yang dibuat berdasarkan perubahan kata yang mungkin pada tata bahasa bahasa Jepang. Pada sisi klien, dibahas sistem yang diharapkan memiliki antarmuka yang semirip mungkin dengan IME desktop.

BAB IV: PEMBAHASAN

Bab ini menjelaskan hasil jadi program. Termasuk di dalamnya adalah contoh eksekusi program dan analisis mengenai kekurangan program.

BAB V: PENUTUP

Bab ini memuat kesimpulan dari implementasi sistem dan saran untuk pengembangan selanjutnya.

BAB II

LANDASAN TEORI

2.1 Sistem Tulis Bahasa Jepang

Pada bahasa Jepang tertulis modern, digunakan 5 jenis simbol (<http://en.wikipedia.org>) yaitu:

- *Hiragana*, yang tiap simbolnya melambangkan silabel
- *Katakana*, melambangkan silabel yang sama seperti *hiragana* (*hiragana* dan *katakana* secara kolektif disebut *kana*)
- *Kanji*, logogram yang berasal dari China
- Alfabet Latin (a-z, A-Z)
- Bilangan Arab (0-9)

Huruf latin hanya digunakan untuk kasus-kasus tertentu, misalnya pada singkatan seperti “CD” dan “DVD”. Bilangan Arab digunakan seperti penggunaannya di Indonesia maupun dunia barat. *Hiragana*, *katakana*, dan *kanji* adalah simbol yang paling banyak muncul pada bahasa Jepang tertulis dan akan dibahas lebih lanjut.

2.1.1 Hiragana

Hiragana merupakan karakter fonetis yang tiap simbolnya melambangkan silabel (suku kata) (Seeley, 1991). Contohnya adalah ひ yang melambangkan silabel “hi”. Ini bisa dikontraskan dengan alfabet Latin pada bahasa Indonesia, yang memerlukan gabungan huruf “h” dan “i” untuk melambangkan bunyi yang

sama.

2.1.1.1 Simbol-Simbol Hiragana

Simbol-simbol dasar *hiragana* melambangkan vokal (misal あ “a”), konsonan+vokal (misal き “ki”), atau n (ん). Suara yang rumit seperti “prak” dan “tik” tidak dikenal pada bahasa Jepang. Daftar simbol dasarnya adalah sebagai berikut:

Tabel 2.1 Hiragana Dasar

あ a	い i	う u	え e	お o
か ka	き ki	く ku	け ke	こ ko
さ sa	し shi	す su	せ se	そ so
た ta	ち chi	つ tsu	て te	と to
な na	に ni	ぬ nu	ね ne	の no
は ha	ひ hi	ふ fu	へ he	ほ ho
ま ma	み mi	む mu	め me	も mo
や ya		ゆ yu		よ yo
ら ra	り ri	る ru	れ re	ろ ro
わ wa	ゐ wi*		ゑ we*	を wo
				ん n

* ゐ (wi) dan ゑ (we) adalah *hiragana* kuno yang saat ini hampir tidak digunakan lagi.

Contoh kata yang bisa ditulis dengan *hiragana* yang telah dijelaskan:

- くるま (kuruma, mobil)
- えいえん (eien, keabadian)
- あう (au, bertemu)

- たなか (tanaka, nama keluarga yang umum di Jepang)
- あんと (anto, transliterasi dari nama orang Indonesia “Anto”)

Beberapa baris pada tabel di atas dapat diberi tanda *dakuten* (゛) untuk mengubah suara $k \rightarrow g$, $s \rightarrow z$, $t \rightarrow d$, dan $h \rightarrow b$. Daftar perubahannya adalah sebagai berikut:

Tabel 2.2 Hiragana dengan Dakuten

Suara Asli	Suara baru				
k	が <i>ga</i>	ぎ <i>gi</i>	ぐ <i>gu</i>	げ <i>ge</i>	ご <i>go</i>
s	ざ <i>za</i>	じ <i>ji</i>	ず <i>zu</i>	ぜ <i>ze</i>	ぞ <i>zo</i>
t	だ <i>da</i>	ぢ (<i>ji</i>)*	づ (<i>zu</i>)*	で <i>de</i>	ど <i>do</i>
h	ば <i>ba</i>	び <i>bi</i>	ぶ <i>bu</i>	べ <i>be</i>	ぼ <i>bo</i>

* Untuk menuliskan bunyi “ji”, pada umumnya yang digunakan adalah じ bukan ぢ. Untuk bunyi “zu”, pada umumnya yang digunakan adalah ず bukan づ. Sebagai contoh, penulisan “kizu” (luka) menggunakan *hiragana* adalah きず, bukan きづ. ぢ dan づ hanya digunakan saat terjadi perubahan suara dalam penggabungan kata. Sebagai contoh, “kanzume” (makanan kaleng) berasal dari “kan” (かん, kaleng) digabung dengan “tsume” (つめ, pengisian). Karena “zume” pada “kanzume” berasal dari “tsume” (つめ) yang berubah suara, maka penulisan “kanzume” adalah かんづめ.

Contoh kata yang bisa ditulis dengan *hiragana* yang telah dijelaskan:

- すうがく (suugaku, matematika)

- りんご (ringo, apel)
- ばか (baka, bodoh)

Ada juga tanda *handakuten* (°) yang mengubah suara h→p:

Tabel 2.3 Hiragana dengan Handakuten

ぱ <i>pa</i>	ぴ <i>pi</i>	ぷ <i>pu</i>	ぺ <i>pe</i>	ぽ <i>po</i>
-------------	-------------	-------------	-------------	-------------

Contoh kata yang bisa ditulis dengan *hiragana* yang telah dijelaskan:

- えんぴつ (enpitsu, pensil)
- しんぴ (shinpi, misteri)
- ぺだる (pedaru, pedal)

Silabel baru bisa dibentuk dengan menyisipkan suara “y” ke dalam struktur konsonan+vokal. Contohnya adalah suara “kya”. Suara seperti ini disebut *youon*. Dengan *hiragana*, “kya” ditulis menggunakan き “ki” yang diikuti oleh *hiragana* kecil や “ya”. Hasilnya adalah きや “kya”. Bandingkan dengan penggunaan *hiragana* normal や “ya” yang menghasilkan きや “kiya”.

Terdapat *hiragana* kecil や “ya”, ゆ “yu”, dan よ “yo” untuk membentuk suara yang disisipi “y” tersebut. Suara yang dapat dibentuk adalah sebagai berikut:

Tabel 2.4 Hiragana dengan Youon

	や ya	ゆ yu	よ yo
き ki	きや <i>kya</i>	きゆ <i>kyu</i>	きよ <i>kyo</i>
し shi	しや <i>sha</i>	しゆ <i>shu</i>	しよ <i>sho</i>
ち chi	ちや <i>cha</i>	ちゆ <i>chu</i>	ちよ <i>cho</i>
に ni	にや <i>nya</i>	にゆ <i>nyu</i>	によ <i>nyo</i>

ひ <i>hi</i>	ひゃ <i>hya</i>	ひゅ <i>hyu</i>	ひょ <i>hyo</i>
み <i>mi</i>	みゃ <i>mya</i>	みゅ <i>myu</i>	みょ <i>myo</i>
り <i>ri</i>	りゃ <i>rya</i>	りゅ <i>ryu</i>	りょ <i>ryo</i>
ぎ <i>gi</i>	ぎゃ <i>gya</i>	ぎゅ <i>gyu</i>	ぎょ <i>gyo</i>
じ <i>ji</i>	じゃ <i>ja</i>	じゅ <i>ju</i>	じょ <i>jo</i>
び <i>bi</i>	びゃ <i>bya</i>	びゅ <i>byu</i>	びょ <i>byo</i>
ぴ <i>pi</i>	ぴゃ <i>pya</i>	ぴゅ <i>pyu</i>	ぴょ <i>pyo</i>

Contoh kata yang bisa ditulis dengan *hiragana* yang telah dijelaskan:

- きよねん (kyonen, tahun lalu)
- びょうき (byouki, penyakit)
- げきじょ (gekijo, teater)

“tsu” kecil (っ) atau *sokuon* digunakan untuk menggandakan suara konsonan yang muncul setelahnya. Sebagai contoh, きて dibaca “kite” sedangkan きてって dibaca “kitte”. *Sokuon* tidak bisa muncul sebelum “a”, “i”, “u”, “e”, maupun “o”. *Sokuon* bisa diletakkan di akhir kalimat yang berarti bahwa suara terakhir berhenti tiba-tiba (menandakan ekspresi seperti marah dan kaget).

Contoh kata yang bisa ditulis dengan *hiragana* yang telah dijelaskan:

- きてって (kitte, perangko)
- ざっし (zasshi, majalah)
- らっぱ (rappa, terompet)

Dakuten juga dapat mengubah う “u” menjadi っ “vu”, namun ini adalah tambahan modern untuk mengakomodasi suara “vu” dari luar negeri. Terdapat juga hiragana versi kecil yang lain yaitu あ “a”, い “i”, う “u”, え “e”, お “o”, dan わ “wa” yang juga untuk mengakomodasi suara luar negeri. Inilah daftar

konstruksi untuk mengakomodasi suara-suara luar tersebut:

Tabel 2.5 Hiragana untuk Suara Luar

うあ va	うい vi	う vu	うえ ve	うお vo	うや vya	うゆ vyu	うよ vyo
			しえ she				
			じえ je				
			ちえ che				
	すい si						
	ずい zi						
	てい ti	てゆ tu				てゆ tyu	
	でい di	でゆ du				でゆ dyu	
つあ tsa	つい tsi		つえ tse	つお tso			
ふあ fa	ふい fi		ふえ fe	ふお fo		ふゆ fyu	
			いえ ye				
	うい wi		うえ we	うお wo			
くわ kwa	くあ kwa	くい kwi	くえ kwe	くお kwo			
ぐわ gwa	ぐあ gwa	ぐい gwi	ぐえ gwe	ぐお gwo			

Contoh kata yang bisa ditulis dengan *hiragana* yang telah dijelaskan:

- うあいおりん (vaiorin, biola)
- ふあっしょん (fasshon, fesyen)
- ふいんらんど (finrando, Finlandia)

2.1.1.2 Penggunaan Hiragana

Karena *hiragana* telah mengakomodasi semua suara yang mungkin pada bahasa Jepang, sebetulnya kalimat lengkap bahasa Jepang bisa saja ditulis hanya dengan *hiragana* (beserta tanda baca seperti titik). Contohnya adalah kalimat berikut:

らどくりふ、まらそん ごりん だいひょう に いちまん めえとる
 しゅつじょう にも ふくみ。
 Radokurifu, marason gorin daihyou ni ichiman meetoru shutsujou ni mo fukumi
 Radcliffe, peserta maraton Olimpiade, juga akan tampil pada kontes 10,000 m.

Namun sebagaimana bahasa Indonesia tidak hanya menggunakan satu set simbol saja (misal huruf kecil saja), kalimat pada bahasa Jepang pada umumnya tidak hanya terdiri dari *hiragana*. Seperti sudah ditulis sebelumnya, angka lebih sering ditulis menggunakan simbol 0-9 dan singkatan-singkatan dari luar negeri seperti CD dan DVD tetap ditulis sebagaimana asalnya. Selain itu masih ada *katakana* dan *kanji*.

Penggunaan khusus *hiragana* mencakup:

- Kata yang tidak memiliki *kanji*, termasuk partikel seperti から “kara” (dari) dan に “ni” (partikel penanda target).
- Kata yang *kanjinya* tidak diketahui penulis, tidak diharapkan untuk diketahui target pembaca, atau terlalu formal untuk keperluan penulisan tersebut.
- Kata yang memiliki *kanji* namun dalam bahasa Jepang modern hampir selalu ditulis menggunakan *hiragana*. Contohnya adalah する

“suru” (melakukan).

- Perbedaan bacaan pada *kanji* yang memiliki lebih dari satu cara membaca. Sebagai contoh, 一 dan 一 つ diawali *kanji* yang sama, namun yang pertama dibaca “ichi” (satu) karena tidak disertai *hiragana* apapun dan yang kedua dibaca “hitotsu” (satu) karena diakhiri *hiragana* “tsu”. Dalam penggunaan ini, *hiragana*-nya disebut sebagai *okurigana*.
- Akhiran infleksional dari verba (kata kerja) dan adjektiva (kata sifat). Sebagai contoh, “tobu” (terbang) jika ditulis menggunakan *kanji* adalah 飛ぶ yaitu *kanji* terbang (飛) ditambah *hiragana* ぶ “bu”. Alasannya adalah karena “bu” pada akhir kata tersebut bisa berubah menjadi bentuk lain, misalnya menjadi “banai” pada bentuk *negatif*. Dalam penggunaan ini, *hiragana*-nya juga disebut sebagai *okurigana*.

2.1.2 Katakana

Katakana melambangkan silabel sama seperti *hiragana* (Seeley, 2001). Untuk setiap *hiragana*, ada *katakana* padanannya. Contohnya silabel “a” pada *hiragana* adalah あ dan pada *katakana* adalah ア. Ini seperti pada alfabet Latin, di mana tiap huruf kecil memiliki huruf besar padanannya.

2.1.2.1 Simbol-Simbol Katakana

Dari segi visual, ciri *katakana* adalah guratannya yang lurus pendek seperti pada 一 “e” dan terdapatnya sudut-sudut yang lancip seperti pada ㇇ “fu”. Karena

penjelasan fonologis pada *hiragana* juga berlaku untuk *katanana*, di bagian ini pembahasan tersebut tidak akan diulang. Inilah tabel simbol-simbol *katakana*:

Tabel 2.6 Katakana Dasar

ア <i>a</i>	イ <i>i</i>	ウ <i>u</i>	エ <i>e</i>	オ <i>o</i>
カ <i>ka</i>	キ <i>ki</i>	ク <i>ku</i>	ケ <i>ke</i>	コ <i>ko</i>
サ <i>sa</i>	シ <i>shi</i>	ス <i>su</i>	セ <i>se</i>	ソ <i>so</i>
タ <i>ta</i>	チ <i>chi</i>	ツ <i>tsu</i>	テ <i>te</i>	ト <i>to</i>
ナ <i>na</i>	ニ <i>ni</i>	ヌ <i>nu</i>	ネ <i>ne</i>	ノ <i>no</i>
ハ <i>ha</i>	ヒ <i>hi</i>	フ <i>fu</i>	ヘ <i>he</i>	ホ <i>ho</i>
マ <i>ma</i>	ミ <i>mi</i>	ム <i>mu</i>	メ <i>me</i>	モ <i>mo</i>
ヤ <i>ya</i>		ユ <i>yu</i>		ヨ <i>yo</i>
ラ <i>ra</i>	リ <i>ri</i>	ル <i>ru</i>	レ <i>re</i>	ロ <i>ro</i>
ワ <i>wa</i>	ヰ <i>wi</i> *		ヱ <i>we</i> *	ヲ <i>wo</i>
				ン <i>n</i>

* ヰ (*wi*) dan ヱ (*we*) adalah katakana kuno yang saat ini hampir tidak digunakan lagi.

Tabel 2.7 Katakana dengan Dakuten

Suara Asli	Suara baru				
k	ガ <i>ga</i>	ギ <i>gi</i>	グ <i>gu</i>	ゲ <i>ge</i>	ゴ <i>go</i>
s	ザ <i>za</i>	ジ <i>ji</i>	ズ <i>zu</i>	ゼ <i>ze</i>	ゾ <i>zo</i>
t	ダ <i>da</i>	ヂ (<i>ji</i>)*	ヅ (<i>zu</i>)*	デ <i>de</i>	ド <i>do</i>
h	バ <i>ba</i>	ビ <i>bi</i>	ブ <i>bu</i>	ベ <i>be</i>	ボ <i>bo</i>

Tabel 2.8 Katakana dengan Handakuten

パ <i>pa</i>	ピ <i>pi</i>	プ <i>pu</i>	ペ <i>pe</i>	ポ <i>po</i>
-------------	-------------	-------------	-------------	-------------

Tabel 2.9 Katakana dengan Youon

	ヤ <i>ya</i>	ユ <i>yu</i>	ヨ <i>yo</i>
キ <i>ki</i>	キヤ <i>kya</i>	キユ <i>kyu</i>	キヨ <i>kyo</i>
シ <i>shi</i>	シヤ <i>sha</i>	シユ <i>shu</i>	シヨ <i>sho</i>
チ <i>chi</i>	チャ <i>cha</i>	チュ <i>chu</i>	チヨ <i>cho</i>
ニ <i>ni</i>	ニヤ <i>nya</i>	ニユ <i>nyu</i>	ニヨ <i>nyo</i>
ヒ <i>hi</i>	ヒヤ <i>hya</i>	ヒユ <i>hyu</i>	ヒヨ <i>hyo</i>
ミ <i>mi</i>	ミヤ <i>mya</i>	ミユ <i>myu</i>	ミヨ <i>myo</i>
リ <i>ri</i>	リヤ <i>rya</i>	リユ <i>ryu</i>	リヨ <i>ryo</i>
ギ <i>gi</i>	ギヤ <i>gya</i>	ギユ <i>gyu</i>	ギヨ <i>gyo</i>
ジ <i>ji</i>	ジヤ <i>ja</i>	ジユ <i>ju</i>	ジヨ <i>jo</i>
ビ <i>bi</i>	ビヤ <i>bya</i>	ビユ <i>byu</i>	ビヨ <i>byo</i>
ピ <i>pi</i>	ピヤ <i>pya</i>	ピユ <i>pyu</i>	ピヨ <i>pyo</i>

Tabel 2.10 Katakana untuk Suara Luar

ヴァ <i>va</i>	ヴィ <i>vi</i>	ヴ <i>vu</i>	ヴェ <i>ve</i>	ヴォ <i>vo</i>	ヴァ <i>vya</i>	ヴュ <i>vyu</i>	ヴョ <i>vyo</i>
			シェ <i>she</i>				
			ジェ <i>je</i>				
			チェ <i>che</i>				
	スイ <i>si</i>						
	ズイ <i>zi</i>						
	テイ <i>ti</i>	トウ <i>tu</i>				テュ <i>tyu</i>	
	デイ <i>di</i>	ドウ <i>du</i>				デュ <i>dyu</i>	
ツァ <i>tsa</i>	ツイ <i>tsi</i>		ツェ <i>tse</i>	ツォ <i>tso</i>			
ファ <i>fa</i>	フィ <i>fi</i>		フェ <i>fe</i>	フォ <i>fo</i>		フュ <i>fyu</i>	

			イエ ye				
	ウイ wi		ウエ we	ウオ wo			
クワ kwa	クアイ kwi		クエ kwe	クオ kwo			
グワ gwa	グアイ gwi		グエ gwe	グオ gwo			

Terdapat juga simbol pemanjang vokal yaitu ー (*chouon*). Simbol tersebut akan memanjangkan vokal yang muncul sebelumnya. Contohnya adalah レポ^ート yang dibaca *repooto* (report, laporan).

2.1.2.2 Penggunaan Katakana

Kata-kata serapan dari luar negeri pada umumnya ditulis menggunakan *katakana*. Contohnya, televisi di bahasa Jepang adalah “tere^{bi}” dan pada umumnya ditulis menggunakan *katakana* テレビ, bukannya *hiragana* てれび. Beberapa contoh kata serapan lain:

- ウイルス (*uirusu*, virus)
- アクション (*akushon*, aksi (dari bahasa Inggris *action*))
- バイト (*baito*, pekerjaan (dari bahasa Jerman *Arbeit*))

Di bagian sebelumnya (bab 2.1.1.1), diberikan penulisan “pedaru” (*pedal*), “fasshon” (*fashion*), dan “vaiorin” (*biola*) menggunakan *hiragana*. Pada umumnya kata-kata tersebut ditulis menggunakan *katakana* karena merupakan kata serapan (secara berurutan ペダル, ファッション, dan ヴァイオリン).

Nama luar negeri, misalnya nama tempat dan nama orang, pada umumnya

juga menggunakan *katakana*. Contohnya adalah:

- インドネシア (indoneshia, Indonesia)
- ジャカルタ (jakaruta, Jakarta)
- トム (tomu, Tom)
- パイジョ (paijo, Paijo)

Di bagian sebelumnya (bab 2.1.1.1), diberikan penulisan nama orang Indonesia “Anto” menggunakan *hiragana*. Pada umumnya, nama tersebut akan ditulis menggunakan *katakana* yaitu アント.

Onomatopia, yaitu kata yang melambangkan suatu bunyi, sering ditulis menggunakan *katakana*. Contohnya:

- ワンワン (wan wan, guk guk)
- ピンポン (pin pon, ding dong)
- ニャニャ (nya nya, meong meong)

Istilah teknis seperti nama spesies juga pada umumnya ditulis menggunakan *katakana*. Contohnya:

- スミレ (sumire, suatu spesies bunga Violet)
- イルカ (iruka, lumba-lumba)
- ハネ (hane, nama suatu gerakan pada permainan papan igo)

Katakana dapat juga digunakan untuk memberi tekanan pada kata-kata yang biasanya ditulis dengan *hiragana* atau *kanji*. Ini sebagaimana penggunaan huruf besar pada bahasa Indonesia, misalnya “DISKON”.

2.1.3 Kanji

Kanji merupakan logogram yang berasal dari China (Seeley, 2001). Tiap simbol *kanji* menyatakan benda atau ide tertentu. Contohnya adalah 母 “haha” yang berarti “ibu” dan 木 “ki” yang berarti “pohon”. Terdapat puluhan ribu *kanji*, namun kebanyakan adalah *kanji* kuno yang tidak dipakai lagi. Pada bahasa Jepang modern, *kanji* yang umum digunakan berjumlah sekitar 2000.

Kata-kata tertentu dapat ditulis hanya menggunakan *kanji*. Sebagai contoh, pada bahasa Jepang “te” berarti “tangan”. Jika ingin menuliskannya dengan *kanji*, maka cukup menuliskan *kanji* untuk tangan yaitu 手. Contoh lain adalah “kokoro” (hati) yang menggunakan *kanji* ditulis sebagai 心.

Namun terdapat juga kata-kata yang penulisan *kanji*-nya selalu disertai *hiragana*. Salah satu alasannya adalah karena *kanji* tersebut memiliki lebih dari satu cara membaca. Sebagai contoh, *kanji* 一 yang berarti “satu” terkait dengan kata “ichi” maupun “hitotsu”. Di sini, keberadaan *hiragana* digunakan untuk mengetahui cara membacanya. Jika hanya ditulis 一 tanpa *hiragana*, maka cara membacanya adalah “ichi”. Jika ditulis 一 つ (*kanji* “satu” ditambah *hiragana* つ “tsu”) maka cara membacanya adalah “hitotsu”. Sebagai contoh lain, *kanji* 教 terkait dengan kata “oshieru” (mengajar) maupun “osowaru” (diajar). 教える (*kanji* “ajar” ditambah *hiragana* える “eru”) berarti bacaannya adalah “oshieru”. 教わる (*kanji* “ajar” ditambah *hiragana* わる “waru”) berarti cara membacanya

adalah “osowaru”.

Alasan lain penyertaan *hiragana* adalah untuk memberitahu bentuk katanya. Suatu kata, misalnya “makan”, bisa berada pada bentuk dasar (taberu), bentuk negatif (tabenai, tidak makan), bentuk lampau (tabeta, telah makan), dan berbagai bentuk lainnya. Yang membedakan bentuk-bentuk tersebut adalah akhir katanya. Pada semua kasus tersebut, *kanji* yang digunakan adalah *kanji* untuk makan yaitu 食 namun disertai akhiran *hiragana* yang sesuai dengan bentuknya. “taberu” ditulis 食べる (diakhiri “beru”), “tabenai” ditulis 食べない (diakhiri “benai”), dst.

Suatu kata, jika ditulis menggunakan *kanji*, dapat juga membutuhkan lebih dari satu *kanji*. “*gakusei*” yang berarti “murid” jika ditulis menggunakan *kanji* adalah 学生 yang terdiri dari *kanji* 学 (belajar) dan 生 (hidup).

Sebagian besar *kanji* memiliki lebih dari satu cara baca. Sebagai contoh, 父 (berarti “ayah”) jika berdiri sendiri dibaca “*chichi*”, dan 母 (berarti “ibu”) jika berdiri sendiri dibaca “*haha*”. Walaupun begitu, jika digabung dengan *kanji* lain 父 dibaca “*fu*” dan 母 dibaca “*bo*”. Contohnya adalah 父母 (berarti “ayah dan ibu”) yang dibaca “*fubo*”.

2.1.3.1 Bacaan on

Bacaan *on* (*on'yomi*) adalah cara orang Jepang membaca *kanji* tersebut berdasarkan cara membaca aslinya di China. Karena suara yang ada di bahasa China lebih bervariasi dibanding suara yang ada di bahasa Jepang, bacaan *on* pada

umumnya hanyalah pendekatan dari cara membaca Chinanya.

Beberapa *kanji* masuk ke Jepang dari berbagai bagian China dan pada waktu yang berbeda, jadi *kanji-kanji* tersebut memiliki lebih dari satu bacaan *on*.

Terdapat juga *kanji* yang dibuat di Jepang dan tidak memiliki bacaan *on*.

Inilah beberapa contoh *kanji* dan bacaan *on*-nya:

Tabel 2.11 Contoh Bacaan On Beberapa Kanji

Kanji	Arti	Bacaan on
明	terang	mei, myou, min
行	pergi	gyou, kou, an
強	kuat	kyou, gou
医	obat	i
変	perubahan	hen
辻	persimpangan	-

2.1.3.2 Bacaan kun

Bacaan *kun* (*kun'yomi*) adalah bacaan yang berdasarkan kata bahasa Jepang asli yang mendekati arti dari simbol Chinanya. Sebagaimana dengan bacaan *on*, *kanji* tertentu memiliki lebih dari satu bacaan *kun* dan *kanji* tertentu tidak memiliki bacaan *kun*.

Sebagai contoh, *kanji* untuk air (水) memiliki bacaan *on* “sui”. Namun orang Jepang sudah memiliki kata untuk air yaitu “mizu”. Maka “mizu” adalah bacaan *kun* untuk *kanji* air tersebut. Namun *kanji* 寸 yang menyatakan satuan panjang dari China tidak dikenal orang Jepang sebelumnya. Jadi 寸 hanya memiliki bacaan *on* yaitu “sun”.

Seringkali, banyak *kanji* digunakan untuk mencakup suatu kata di bahasa

Jepang. Biasanya tiap kanji menopang penggunaan berbeda dari kata tersebut. Contohnya, kata あつい “atsui” (panas) jika ditulis 暑い berarti mengacu pada cuaca yang panas, sedangkan jika ditulis 熱い berarti mengacu pada benda yang panas (misal tubuh).

Inilah contoh beberapa *kanji* dan bacaan *kun*-nya. Tanda minus (-) digunakan untuk mendandai batas *okurigana*.

Tabel 2.12 Contoh Bacaan Kun Beberapa Kanji

Kanji	Arti	Bacaan kun
間	waktu	aida, ma
止	berhenti	ya-mu, to-maru, todo-maru, to-meru, ya-meru, dan yang lainnya
鳥	burung	tori
美	indah	utsuku-shii
仙	pertapa	-

2.1.3.3 Bacaan nanori

Bacaan *nanori* adalah bacaan khusus yang hanya digunakan untuk nama tempat atau orang. Sebagai contoh, pada nama 辻希美 (Tsuji Nozomi) *kanji* 希 dibaca “nozo” yang merupakan bacaan *nanori*-nya.

2.1.3.4 Bacaan yang digunakan

Suatu *kanji* bisa memiliki banyak bacaan *on* dan *kun*, dan tidak ada cara yang pasti (kecuali melihat kamus) untuk mengetahui bacaan yang benar. Namun ada beberapa panduan yang berlaku bagi banyak kasus.

Secara umum, *kanji* yang digabung dengan *kanji* lain dibaca menggunakan

bacaan *on*-nya. Contohnya adalah 先生 “sensei” (guru), 残念 “zannen” (penyesalan), dan 平和 “heiwa” (kedamaian).

Kanji yang berdiri sendiri atau hanya bersebelahan dengan hiragana pada umumnya dibaca menggunakan bacaan *kun*-nya. Contohnya adalah 風 “kaze” (angin), 青い “aoi” (biru), dan 動く “ugoku” (bergerak).

Dua aturan di atas memiliki banyak pengecualian. Banyak gabungan *kanji* yang dibaca menggunakan bacaan *kun*-nya. Contohnya adalah 手紙 “tegami” (surat) dan 神風 “kamikaze” (angin agung). Gabungan *kanji* tersebut juga bisa menggunakan *okiragana*, misalnya 折り紙 “origami” (seni lipat kertas) (walaupun banyak juga yang bisa ditulis tanpa *okirigana*, misalnya 折紙 juga merupakan penulisan untuk “origami”).

Di lain pihak, beberapa *kanji* dibaca menggunakan bacaan *on*-nya walaupun berdiri sendiri. Contohnya adalah 愛 “ai” (cinta) dan 点 “ten” (titik). Namun biasanya *kanji-kanji* tersebut tidak memiliki bacaan *kun* sehingga tidak ada ambiguitas.

Cara membaca dipersulit karena suatu *kanji* bisa memiliki lebih dari satu bacaan *on* maupun *kun*. Contohnya 先生 “sensei” (guru) dan 一生 “isshou” (seumur hidup) yang sama-sama diakhiri *kanji* 生 namun dengan bacaan yang berbeda. Sebagai contoh lain, 間 (rentang waktu) memiliki dua bacaan *kun* yaitu “aida” dan “ma”.

Terdapat juga bacaan *gikun* yaitu bacaan dari gabungan *kanji* yang tidak berhubungan dengan bacaan *kun* maupun *on*-nya. Contohnya 明日 (besok) bisa

dibaca “asu” maupun “ashita”, keduanya tidak diturunkan dari bacaan *on* maupun bacaan *kun*-nya. Contoh lainnya adalah 今朝 (pagi ini) yang dibaca “kesa”.

Kadangkala *kanji* digunakan hanya karena keperluan fonetis. Sebagai contoh, penulisan kuno dari “amerika” adalah 亜米利加. Dalam hal ini, kanji-kanji tersebut digunakan karena memiliki bacaan berturut-turut “a”, “me”, “ri”, dan “ka”. Arti masing-masing kanjinya secara historis sama sekali tidak berhubungan dengan Amerika. Penggunaan seperti ini disebut *ateji*.

Beberapa nama tempat seperti Tokyo (東京, toukyou) dan Jepang (日本, nihon atau nippon) dibaca menggunakan bacaan *on*-nya. Namun sebagian besar nama tempat di Jepang dibaca menggunakan bacaan *kun*-nya, seperti 大阪 (oosaka), 岩手 (iwate), dan 山口 (yamaguchi). Nama keluarga biasanya juga dibaca menggunakan bacaan *kun*, seperti 高橋 (takahashi), 田中 (tanaka), dan 亀井 (kamei). Nama panggilan biasanya dibaca menggunakan gabungan bacaan *on*, *kun*, dan *nanori*. Contohnya adalah 美貴 (miki, on-on), 小春 (koharu, kun-kun), dan 希美 (nozomi, nanori-on).

2.2 Transliterasi Bahasa Jepang menggunakan Huruf Latin

Transliterasi kata bahasa Jepang menggunakan huruf latin disebut *romaji* (ローマ字, “roomaji”) (Seeley, 2001). Alasan penggunaan *romaji* misalnya untuk tanda jalan agar dimengerti orang asing, penulisan nama orang atau perusahaan untuk keperluan internasional, dan kamus untuk pelajar bahasa Jepang.

Terdapat beberapa sistem romanisasi, dan yang paling utama adalah sistem Hepburn, Nihon-shiki, dan Kunrei-shiki. Romanisasi pada tabel *hiragana* dan

katakana di bagian 2.1.1 dan 2.1.2 berdasarkan pada Hepburn. Perbedaannya yang relevan untuk pengembangan IME adalah sebagai berikut:

- Konsistensi dengan fonologi Jepang: Nihon-shiki dan Kunrei-shiki mengikuti fonologi Jepang, sedangkan Hepburn memilih untuk memberikan petunjuk terbaik tentang cara pengucapan suatu silabel bahasa Jepang bagi pengguna bahasa Inggris. Sebagai contoh, fonologi bahasa Jepang menganggap kana た, ち, つ, て, dan と sebagai kana dengan konsonan 't' dan vokalnya masing-masing '-a', '-i', '-u', '-e', '-o'. Pengelompokan tersebut penting dalam formulasi berbagai aturan tata bahasa Jepang. Nihon-shiki mengikutinya dan menuliskannya sebagai 'ta', 'ti', 'tu', 'te', 'to'. Di lain pihak Hepburn menuliskannya 'ti' sebagai 'chi' dan 'tu' sebagai 'tsu', mempertimbangkan bagaimana suaranya sebenarnya diucapkan. Kunrei-shiki berusaha mengikuti fonologi Jepang namun memiliki beberapa perkecualian¹. Inilah daftar kana yang transliterasinya berbeda karena faktor tersebut:

Tabel 2.13 Perbedaan konsistensi fonologis Hepburn, Nihon-shiki, dan Kunrei-shiki

Kana	Hepburn	Nihon-shiki	Kunrei-shiki
し/シ	shi	si	si
ち/チ	chi	ti	ti
つ/ツ	tsu	tu	tu
ふ/フ	fu	hu	hu

1 Jika pada penggabungan 2 kata terjadi perubahan suara, misalnya *かな* “kana” + *つかい* “tukai” menjadi *かなづかい*, Kunrei-shiki memilih “kanazukai” yang sesuai bunyi modernnya, bukannya “kanadukai” yang sesuai dengan penulisan *kana*-nya

じ/ジ	ji	zi	zi
ぢ/ヂ	ji	di	di
づ/ヅ	zu	du	du
ゐ/ヰ	wi	wi	i
ゑ/ヱ	we	we	e
じ/ジ	ji	zi	zi
ぢ/ヂ	ji	di	di
づ/ヅ	zu	du	du
しゃ/シャ	sha	sya	sya
しゅ/シュ	shu	syu	syu
しょ/ショ	sho	syo	syo
ちゃ/チャ	cha	tya	tya
ちゅ/チュ	chu	tyu	tyu
ちょ/チョ	cho	tyo	tyo
じゃ/ジャ	ja	zya	zya
じゅ/ジュ	ju	zyu	zyu
じょ/ジョ	jo	zyo	zyo
ぢゃ/ヂャ	ja	dya	dya
ぢゅ/ヂュ	ju	dyu	dyu
ぢょ/ヂョ	jo	dyo	dyo

Sebagai contoh, ふじさん (gunung Fuji) akan menjadi “fujisan” pada Heburn dan “huzisan” pada Nihon-shiki dan Kunrei-shiki.

- Perlakuan khusus untuk partikel: Pada bahasa Jepang, saat は “ha”, へ “he”, dan を “wo” digunakan sebagai partikel, suaranya berubah menjadi “wa”, “e”, dan “o”. Nihon-shiki tidak mempedulikan hal tersebut dan romanisasinya tetap “ha”, “he”, dan “wo”. Hepburn dan Kunrei-shiki mengubah romanisasinya menjadi “wa”, “e”, dan “o”.

Sebagai contoh, *これはくるま* (ini mobil) akan menjadi “kore wa kuruma” pada Hepburn dan Kunrei-shiki dan “kore ha kuruma” pada Nihon-shiki.

- Silabel n (ん): Pada Nihon-shiki dan Kunrei-shiki, ん ditulis sebagai “n” (“n” diikuti apostrof) sebelum vokal dan “y”, dan ditulis sebagai “n” pada kasus lainnya. Sistem Hepburn sebetulnya terdiri lagi atas Hepburn tradisional, revisi, dan modifikasi yang menangani silabel ん dengan cara yang berbeda-beda. Pada Hepburn tradisional, ん ditulis sebagai “m” sebelum konsonan “b”, “m”, dan “p”. Untuk kasus lainnya digunakan sistem yang sama seperti Nihon-shiki/Kunrei-shiki. Hepburn revisi menggunakan sistem yang sama persis dengan Nihon-shiki/Kunrei-shiki. Hepburn modifikasi menuliskan ん sebagai “n” dengan tanda makron (garis di atasnya).

Sebagai contoh, *しんぶん* (koran) ditulis sebagai “shinbun” pada Hepburn revisi, Nihon-shiki, dan Kunrei-shiki dan sebagai “shimbun” pada Hepburn tradisional. *きんえん* (dilarang merokok) ditulis sebagai “kin'en” pada semua sistem (kecuali Hepburn modifikasi). Penggunaan apostrof setelah “n” adalah untuk menghindari potensi disambiguasi, misalnya dengan *きねん* “kinen” (perayaan).

- Konsonan ganda: Nihon-shiki dan Kunrei-shiki menandainya dengan menggandakan konsonan yang muncul setelah つ tanpa perkecualian. Pada Hepburn perkecualiannya adalah “ch”→”tch” , bukannya “cch”.

Sebagai contoh らっぱ (terompet) menjadi “rappa” pada semua sistem sedangkan “ぼっちゃん (tuan muda) menjadi “bottan” pada Nihon-shiki/Kunrei-shiki dan “botchan” pada Hepburn.

2.3 Modifikasi Kata Dasar pada Bahasa Jepang

Tidak seperti bahasa Indonesia, pada bahasa Jepang infleksi (perubahan bentuk kata) digunakan jauh lebih ekstensif. Bentuk dasar suatu kata dapat berubah ke berbagai macam bentuk lain (Kaiser, 2001). Untuk tujuan yang sama, bahasa Indonesia menggunakan kata-kata tambahan. Inilah contohnya untuk kata “tobu” (terbang):

Tabel 2.14 Padanan bahasa Indonesia beberapa infleksi pada bahasa Jepang

Bentuk	Bahasa Indonesia	Bahasa Jepang
dasar	terbang	tobu
negatif	<u>tidak</u> terbang	tob <u>anai</u>
lampau	<u>telah</u> terbang	tob <u>nda</u>
desideratif	<u>ingin</u> terbang	tob <u>itai</u>
potensial	<u>bisa</u> terbang	tob <u>eru</u>
volisional	<u>ayo</u> terbang	tob <u>ou</u>
sopan	-	tob <u>imasu</u>

Jika ditulis menggunakan *kanji*, *kanji* dasarnya sama. Yang berbeda adalah *hiragana* yang mengakhirinya. Inilah contohnya:

Tabel 2.15 Ilustrasi penulisan “tobu” dan berbagai infleksinya

Romaji	Kanji	Komponen
tobu	飛ぶ	kanji terbang (飛) + hiragana "bu"
tobanai	飛ばない	kanji terbang (飛) + hiragana "banai"

tonda	飛んだ	kanji terbang (飛) + hiragana "nda"
tondeiru	飛んでいる	kanji terbang (飛) + hiragana "ndeiru"
tobitai	飛びたい	kanji terbang (飛) + hiragana "bitai"
toberu	飛べる	kanji terbang (飛) + hiragana "beru"
tobou	飛ぼう	kanji terbang (飛) + hiragana "bou"
tobimasu	飛びます	kanji terbang (飛) + hiragana "bimasu"

Dalam contoh di atas, bagian *kanji* 飛 dibaca "to" pada semua jenis infleksi. Ini karena infleksi pada "tobu" hanya mengubah akhir katanya. Namun terdapat juga infleksi yang mengubah katanya secara keseluruhan. Contohnya adalah "kuru" (datang):

Tabel 2.16 Beberapa infleksi “kuru”

Bahasa Indonesia	Bahasa Jepang	Kanji
datang	kuru	来る
<u>tidak</u> datang	<u>konai</u>	来ない
<u>telah</u> datang	<u>kita</u>	来た

Pada contoh di atas, bagian kanji 来 bisa dibaca "ku", "ko", maupun "ki" tergantung bentuk infleksi yang sedang dipakai.

Infleksi tertentu sangatlah seragam. Sebagai contoh, untuk mengubah verba apapun ke bentuk “kondisional -ba” (misal “makan” ke “jika makan”), yang perlu dilakukan hanyalah mengganti suara vokal “u” di akhir² menjadi “e” dan menambahkan “ba”. Contohnya:

- “taberu” (makan) → “tabereba” (jika makan)
- “tobu” (terbang) → “tobeba” (jika terbang)

2 Semua bentuk dasar kata kerja di bahasa Jepang pasti berakhiran -u (misalnya u, ru, dan nu)

- “suru” (melakukan) → “sureba” (jika melakukan)
- “kuru” (datang) → “kureba” (jika datang)

Terdapat juga infleksi yang lebih rumit karena membutuhkan pengklasifikasian bentuk dasarnya. Contohnya adalah infleksi verba ke bentuk lampainya:

Tabel 2.17 Infleksi lampau berbagai jenis verba

Jenis verba (tidak lengkap)	Contoh kata	Bentuk lampau
<i>ichidan</i>	taberu (makan)	tabeta
<i>godan</i> baris su	hana <u>su</u> (bicara)	hanashita
<i>godan</i> baris ku	aru <u>ku</u> (jalan)	aru <u>ita</u>
<i>godan</i> baris gu	oyogu (berenang)	oyoida
<i>godan</i> baris mu, bu, nu	to <u>bu</u> (terbang)	tonda
<i>godan</i> baris ru, u, tsu	iu (mengatakan)	itta
<i>godan</i> baris ku, perkecualian	iku (pergi)	itta
suru	suru (melakukan)	shita
kuru	kuru (datang)	kita

Hasil suatu infleksi dapat juga diinfleksikan lagi sehingga membentuk rantai infleksi yang panjang. Sebagai contoh, “dulu tidak bisa membaca” pada bahasa Jepang adalah “yomenakatta”, yang pembentukannya adalah:

1. yomu (bentuk dasar, “membaca”)
2. yomeru (bentuk potensial dari “yomu”, “bisa membaca”)
3. yomenai (bentuk negatif dari “yomeru”, “tidak bisa membaca”)
4. yomenakatta (bentuk lampau dari “yomenai”, “dulu tidak bisa membaca”)

Selain infleksi yang mengubah akhir kata, terdapat juga infleksi berupa penambahan awalan. Contohnya adalah penambahan “o-” dan “go-” untuk kata benda guna menunjukkan kesopanan. Contoh kasusnya adalah “shigoto” (pekerjaan) menjadi “oshigoto” dan “houbi” (hadiah) menjadi “gohoubi”.

Infleksi tertentu dapat ditulis menggunakan kanji. Sebagai contoh, pada “motsu” (memiliki) yang diinfleksi menjadi “motteiku”, “iku” bisa ditulis menggunakan hiragana いく maupun kanji 行く.

2.4 Partikel dan Gobi

Pada bahasa Jepang, partikel digunakan untuk menandai fungsi kata yang mendahuluinya (Kaiser, 2001). Sebagai contoh, pada kalimat “hon ga ie ni aru” (buku ada di rumah) terdapat 2 partikel yang semuanya digarisbawahi. Partikel “ga” yang mengikuti “hon” (buku) menandakan bahwa buku adalah subyeknya. Partikel “ni” yang mengikuti “ie” (rumah) menandakan bahwa rumah adalah lokasi keberadaaan.

Gobi adalah akhiran kalimat yang memberikan nuansa tertentu pada kalimatnya (Kim). Sebagai contoh, *gobi* “ne” menghadirkan nuansa meminta persetujuan pendengar atas pernyataan yang baru diberikan. Contoh kalimatnya adalah “fushigi ne” (aneh kan?).

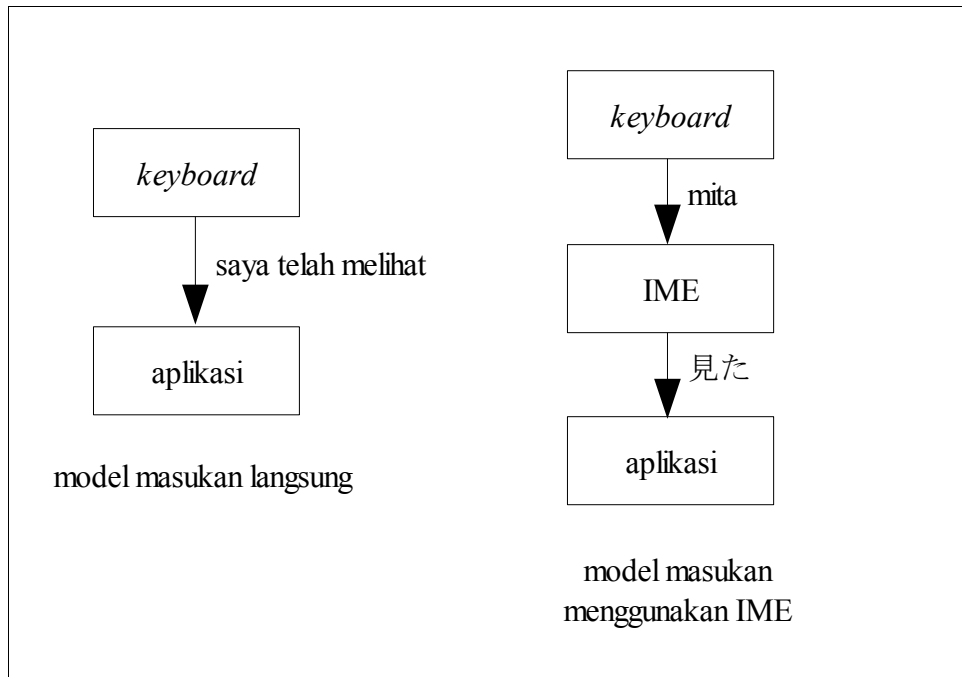
2.5 Input Method Editor (IME)

Bahasa berbasis huruf Latin (misalnya Indonesia, Inggris, dan Perancis)

secara tertulis direpresentasikan dengan kombinasi karakter yang jumlahnya terbatas. Karena jumlah karakternya relatif sedikit, pada bahasa-bahasa tersebut tiap karakter berkorespondensi dengan satu tombol *keyboard* (dengan bantuan tombol modifikasi seperti *shift*).

Di lain pihak, pada bahasa Jepang jumlah karakternya mencapai ribuan, sehingga model korenspondensi satu-satu antara karakter yang ada dengan tombol di *keyboard* tidak layak diterapkan. Untuk memungkinkan pemasukan karakter bahasa Jepang, beberapa pengolah metode masukan (*Input Method Editor*, IME) telah dikembangkan, seperti IME Windows XP (<http://www.microsoft.com>) dan SCIM (<http://www.scim-im.org>) untuk Linux.

IME berada di antara aplikasi dengan pengguna. Karena sifatnya yang merupakan perantara, secara umum pengguna dapat memasukkan karakter yang diinginkan pada berbagai macam aplikasi tanpa perlu modifikasi apapun di pihak aplikasi. Lebih tepatnya, IME bisa diaktifkan jika fokus suatu program berada pada *widget* yang mendukung masukan (misalnya kotak teks). Hal ini diilustrasikan pada Gambar 2.1.



Gambar 2.1 Kedudukan IME pada proses masukan

Pada umumnya IME menyediakan berbagai alternatif cara untuk memasukkan karakter yang diinginkan:

- Masukan berbasis fotenis: Pengguna menuliskan romanisasi dari kata yang diinginkan, lalu IME akan memberikan alternatif kata yang mungkin.
- Pengenalan tulisan tangan: Dengan *mouse*, pengguna menggambar karakter yang diinginkan pada suatu kanvas lalu IME memberikan alternatif karakter yang dideteksinya.
- Pengenalan suara: Pengguna mengucapkan kata yang diinginkan menggunakan mikrofon dan IME memberikan alternatif kata yang dideteksinya.
- Pencarian berdasarkan kriteria: Pengguna memberikan kriteria tertentu misalnya jumlah goresan karakter yang diinginkan dan IME memberikan

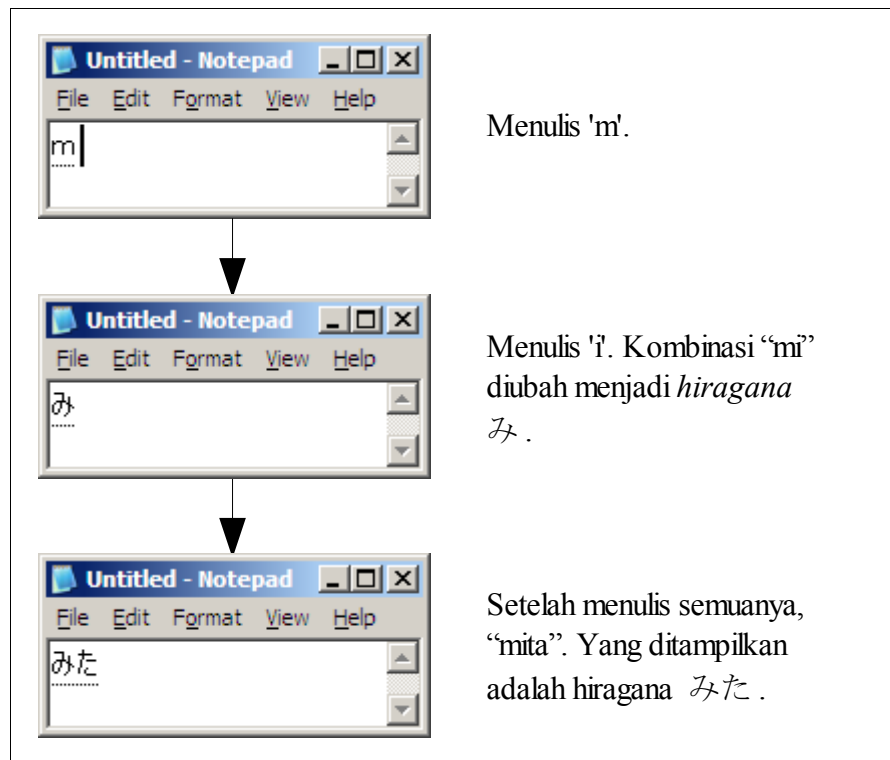
daftar karakter yang memenuhi kriteria tersebut sehingga pengguna dapat melakukan pemilihan lebih lanjut.

Di antara metode-metode tersebut, yang paling populer adalah metode berbasis fonetis, yang akan dibahas lebih lanjut. Karena cara kerja berbagai IME berbasis fonetis relatif sama, hanya akan dicontohkan prinsip kerja salah satu produk saja yaitu IME Windows XP.

2.5.1 IME Fonetis Windows XP

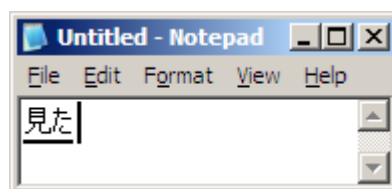
Jika IME telah terinstall, kita bisa memasukkan karakter bahasa Jepang pada *widget* yang mendukung masukan. Langkahnya adalah:

- Berikan fokus pada *widget* yang diinginkan, misalnya pada tempat mengetik aplikasi Notepad atau OpenOffice.org Writer
- Aktifkan IME dengan menekan tombol aktivasinya. Pada Windows XP tombol untuk mengaktifkannya adalah ALT+` (tombol di atas TAB).
- Tuliskan *romaji* dari kata yang diinginkan. Contohnya adalah “mita” (telah melihat). Saat menulis *romaji*-nya, secara waktu nyata akan ditampilkan *hiragana*-nya. Kata yang ditulis juga akan digarisbawahi untuk menandakan bahwa sebetulnya karakter-karakter tersebut belum dikirim ke aplikasi yang bersangkutan. Contoh tampilan setelah menuliskan 'm', lalu 'i', lalu sisanya ('ta') ditunjukkan pada Gambar 2.2:



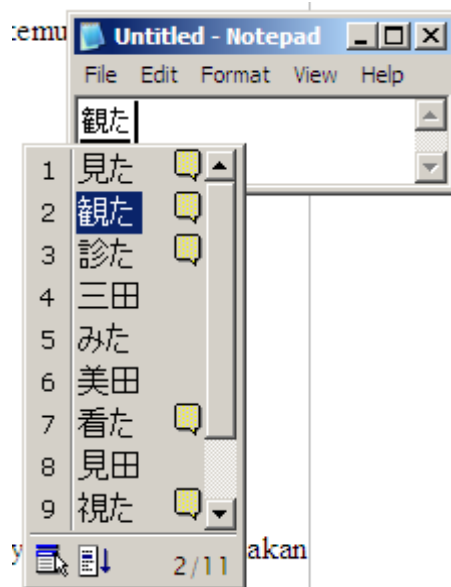
Gambar 2.2 Konversi “mita” pada IME

- Tekan tombol konversi untuk mengubahnya ke *kanji*. Pada Windows XP, tombol konversinya adalah spasi. IME akan memilih kandidat yang dianggap paling mungkin dari kemungkinan-kemungkinan yang ada. Contoh hasil konversi ditunjukkan pada Gambar 2.3:



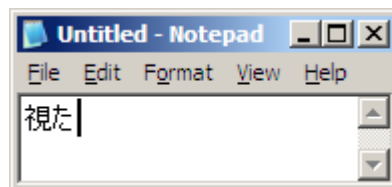
Gambar 2.3 Hasil konversi “mita”

- Jika hasil konversi bukanlah yang diinginkan, tekan lagi tombol konversi untuk memilih kemungkinan berikutnya. Daftar kemungkinan yang ada juga akan ditunjukkan. Ini diperlihatkan di Gambar 2.4:



Gambar 2.4 Tampilan kemungkinan “mita”

- Tekan enter setelah memilih konversi yang diinginkan. Hasilnya akan dikirim ke aplikasi yang bersangkutan, ditunjukkan pada Gambar 2.5:



Gambar 2.5 Hasil konversi “mita” yang telah dikirim ke aplikasi

2.6 Himpunan Karakter dan Pengkodean Bahasa Jepang di Komputer

2.6.1 Himpunan Karakter dan Pengkodean

Himpunan karakter (*character set*) adalah pemetaan satu-satu dari suatu himpunan bilangan bulat ke himpunan simbol tertulis (Elias). Contohnya adalah himpunan karakter fiksional yang memetakan alfabet A ke 1, B ke 2, dan C ke 3 (dan tidak ada karakter lain lagi).

Di lain pihak, pengkodean (*encoding*) adalah cara konkrit menyimpan karakter tersebut ke 0 dan 1 pada memori komputer. Suatu himpunan karakter

bisa memiliki banyak encoding. Menggunakan contoh sebelumnya, A yang berkorespondensi dengan 1 bisa dikodekan menggunakan 8 bit dengan cara yang umum mengkodekan bilangan bulat:

```
00000001
```

Namun dapat juga digunakan pengkodean lain. Pada contoh himpunan karakter sebelumnya, karena hanya terdapat 3 karakter maka mungkin saja digunakan pengkodean 2 bit sehingga karakter A akan dikodekan menjadi:

```
01
```

2.6.2 Himpunan Karakter Bahasa Jepang

Terdapat dua himpunan karakter untuk menulis bahasa Jepang yaitu JIS (Japanese Industrial Standard) dan Unicode. Unicode merupakan himpunan karakter yang berusaha mengakomodasi seluruh bahasa, dan bagian Jepang dari Unicode diturunkan dari standar JIS (Elias).

Dari sudut pandang pemrogram secara umum, yang perlu dikhawatirkan hanyalah himpunan karakter Unicode. Ini karena walaupun teks bahasa Jepang pada berkas bisa disimpan menggunakan pengkodean yang berbasis JIS, saat dibaca ke memori representasinya adalah Unicode pada kebanyakan bahasa pemrograman modern (misalnya `wchar_t` C++, `string` C#, `string` Java, dan `string` JavaScript).

Dalam standar Unicode, titik kode suatu karakter biasa dituliskan menggunakan format `U+XXXX` dengan `XXXX` adalah kodenya dalam heksadesimal.

2.6.2.1 Hiragana pada Unicode

Pada Unicode, Hiragana menduduki titik kode U+3040 sampai U+309F:

Tabel 2.18 Hiragana pada Unicode

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+304x	N/A	あ	い	う	え	お	か	き	く							
U+305x	ぐ	け	こ	さ	し	す	せ	そ	た							
U+306x	だ	ち	っ	つ	て	と	ど	な	ぬ	ね	の	は				
U+307x	ば	び	ひ	ふ	ぶ	へ	べ	ほ	ぼ	ま	み					
U+308x	む	め	も	や	ゆ	よ	ら	り	る	れ	ろ	わ				
U+309x	ゐ	ゑ	を	ん	う*	* ゐ	N/A	N/A	* ゑ	* を	ゝ	ゞ	*			

* Entri yang diberi tanda bintang akan dijelaskan di bawah, karena tidak bisa ditampilkan pada font yang dimiliki penulis

Blok Unicode *hiragana* berisi semua karakter *hiragana* modern, termasuk vokal kecil, *hiragana* untuk youon, *hiragana* kuno “wi” dan “we”, dan “vu” yang jarang dipakai. Semua *hiragana* dengan *dakuten* dan *handakuten* yang umum adalah karakter sendiri yang dapat langsung dipakai. Karakter yang sama dapat dibuat dengan menggunakan *hiragana* dasarnya ditambah karakter *dakuten* dan *handakuten* penggabung (masing-masing U+3099 dan U+309A). U+3095 adalah か “ka” kecil dan U+3096 adalah け “ke” kecil. U+309F adalah digraf より “yori” yang sering dipakai pada teks vertikal.

2.6.2.2 Katakana pada Unicode

Terdapat dua jenis *katakana* pada Unicode yaitu *katakana* lebar penuh

(*fullwidth*) dan *katakana* lebar setengah (*halfwidth*). Ketersediaan dua jenis tersebut karena alasan historis.

Pada awalnya, terdapat himpunan karakter JIS X 0201 untuk keperluan bahasa Jepang. Himpunan karakter tersebut dirancang agar bisa dikodekan dalam 8-bit sehingga tidak mungkin memuat seluruh *hiragana*, *katakana*, dan *kanji*. Yang dipilih untuk disertakan adalah *katakana*. Pada pengkodean tersebut 128 karakter pertama sama dengan ASCII kecuali *backslash* (\) yang diganti Yen (¥) dan tilda (~) yang diganti garis atas (^). *Katakana* diletakkan pada sebagian dari 128 karakter berikutnya. Oleh karenanya, dengan himpunan karakter tersebut *katakana* dapat dikodekan dalam 8-bit. Saat ditampilkan di terminal, *katakana* tersebut memiliki lebar yang sama dengan karakter ASCII lainnya.

Berikutnya, muncul himpunan karakter yang lebih besar agar bisa mengakomodasi *hiragana* dan *kanji* yaitu JIS X 0208. Pada pengkodean karakter tersebut, *hiragana* dan *kanji* membutuhkan lebih dari 1 byte untuk dikodekan, dan penampilannya di terminal memakan tempat dua kali lebih lebar dibanding karakter ASCII untuk mengakomodasi *kanji* yang bentuknya kompleks. Oleh karenanya, karakter-karakter tersebut dinamakan karakter lebar penuh. *Katakana* lebar setengah tetap dipertahankan untuk alasan kompatibilitas, namun karena lebarnya yang setengah membuatnya terlihat berbeda dari karakter lebar penuh, maka dibuat juga titik kode untuk *katakana* dengan lebar penuh. Jadi pada JIS X 0208 terdapat dua jenis *katakana* yaitu *katakana* lebar setengah yang muncul lebih awal dan *katakana* lebar penuh yang jika ditampilkan memiliki lebar yang sama dengan *kanji* dan *hiragana*.

Dua jenis katakana tersebut dibawa ke himpunan karakter Unicode. Kali ini yang berbeda adalah tampilannya, namun keduanya sama-sama memerlukan lebih dari 1 byte pada pengkodean standarnya.

Katakana lebar penuh menduduki titik kode U+30A0 sampai U+30FF:

Tabel 2.19 *Katakana* Lebar Penuh pada Unicode

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+30Ax	N/A	ァ	ア	ィ	イ	ゥ	ウ	ェ	エ	ォ	オ	カ	ガ	キ	ギ	ク
U+30Bx	グ	ケ	ゲ	コ	ゴ	サ	ザ	シ	ジ	ス	ズ	セ	ゼ	ソ	ゾ	タ
U+30Cx	ダ	チ	ヂ	ツ	ツ	ヅ	テ	デ	ト	ド	ナ	ニ	ヌ	ネ	ノ	ハ
U+30Dx	バ	パ	ヒ	ビ	ピ	フ	ブ	プ	ヘ	ベ	ペ	ホ	ボ	ポ	マ	ミ
U+30Ex	ム	メ	モ	ヤ	ヤ	ユ	ユ	ヨ	ヨ	ラ	リ	ル	レ	ロ	ワ	ワ
U+30Fx	ヰ	ヱ	ヲ	ン	ヴ	カ	ケ	ヅ	ヰ	ヱ	ヰ	・	ー	ヽ	ヾ	N/A

Katakana lebar setengah menduduki titik kode U+FF65 sampai U+FF9F:

Tabel 2.20 *Katakana* Lebar Setengah pada Unicode

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+FF6x	・	。	「	」	、	・	ヲ	ァ	ィ	ゥ	ェ	ォ	ヤ	ユ	ヨ	ツ
U+FF7x	ー	ア	ィ	ウ	ェ	オ	カ	キ	ク	ケ	コ	サ	シ	ス	セ	ソ
U+FF8x	タ	チ	ツ	テ	ト	ナ	ニ	ヌ	ネ	ノ	ハ	ヒ	フ	ヘ	ホ	マ
U+FF9x	ミ	ム	メ	モ	ヤ	ユ	ヨ	ラ	リ	ル	レ	ロ	ワ	ン	ゝ	。

2.6.2.3 Kanji pada Unicode

Pada Unicode, *kanji* bahasa Jepang diunifikasi dengan karakter sejenis yang digunakan di China (*Hànzi* tradisional) dan Korea (*Hanja*). Keputusan ini disebut unifikasi Han. Ini berarti bahwa suatu karakter yang secara historis sama namun digambar dengan sedikit variasi pada negara yang berbeda hanya akan diberi satu titik kode. Bagaimana karakter tersebut pada akhirnya akan digambar

di layar akan tergantung dari setting bahasa (misal atribut `lang` HTML) dan *font* yang tersedia.

2.7 Breadth-first Search (BFS)

Breadth-first search (BFS) adalah algoritma pencarian pada graf yang memulai pencarian dari *node* awal dan menjelajahi semua *node* tetangganya. Lalu untuk tiap tetangganya tersebut, seluruh tetangga berikutnya dijelajahi, sampai suatu gol dicapai (Cormen, 2001).

Dari sudut pandang algoritma, seluruh *node* anak yang diperoleh dimasukkan ke dalam antrian FIFO (*First In First Out*).

Pseudocode-nya adalah sebagai berikut:

1. Letakkan *node* awal di antrian
2. Ambil *node* dari awal antrian dan lakukan pemeriksaan. Jika tujuan telah tercapai di *node* ini, hentikan pencarian dan kembalikan hasilnya
3. Jika tidak, cari seluruh anak *node* tersebut yang belum dijelajahi dan masukkan ke akhir antrian.
4. Ulangi langkah 2

Breadth-first search banyak digunakan pada masalah-masalah kecerdasan buatan. Pada masalah-masalah tersebut, selain mencari keadaan yang merupakan solusi, penting juga untuk menyimpan informasi urutan-urutan langkah untuk mendapat solusi tersebut.

2.8 EDICT

EDICT adalah kamus elektronik Jepang-Inggris yang bebas digunakan

selama penggunaannya diakui. Penyusun utamanya adalah Jim Breen dari Universitas Monash, Australia (Breen).

2.8.1 Format

EDICT adalah berkas teks dengan satu entri tiap barisnya. Untuk pengkodean *kana* dan *kanji* digunakan EUC-JP. Semua karakter lain, termasuk spasi, dikodekan dengan ASCII. Baris pertama berisi informasi versi EDICT.

Format baris sisanya adalah:

```
KANJI [KANJI] /Inggris_1/Inggris_2/.../
```

atau (untuk kata yang tidak memiliki kanji)

```
KANA /Inggris_1/.../
```

Tidak dijamin urutan tertentu dalam penyusunan entri-entrinya. Namun pada prakteknya urutannya hampir selalu JIS+alfabetis.

Pada EDICT, infleksi verba dan adjektiva tidak disertakan kecuali untuk idiom tertentu. Sebagai contoh, adverbial (kata keterangan) yang dibentuk dari kata sifat (misalnya 遅く (osoku, dengan lambat) dari 遅い (osoi, lambat)) secara umum tidak disertakan. Perkecualiannya adalah jika kata yang bersangkutan merupakan ekepresi idiomatis seperti いらっしやい (irasshai, selamat datang) Verba ditulis dalam bentuk dasar atau bentuk “kamus”.

20 ribu entri yang merupakan kata yang paling umum digunakan di bahasa Jepang ditandai dengan “(P)” pada akhir entri.

Tiap entri juga ditandai dengan sejumlah penanda (*tag*) kelas kata, yang memberitahu apakah kata yang bersangkutan merupakan verba, adjektiva, atau

yang lainnya. Penanda tersebut spesifik untuk bahasa Jepang. Sebagai contoh, pada tata bahasa Jepang adjektiva (kata sifat) dibagi lagi menjadi adjektiva-i (*keiyoushi*), adjektiva-na (*keiyoudoushi*), dan yang lainnya. Untuk mengakomodasi hal tersebut maka terdapat penanda adj untuk adjektiva-i, adj-na untuk adjektiva-na, dan penanda-penanda lainnya untuk adjektiva yang lain. Inilah daftar penanda yang ada:

Tabel 2.21 Penanda Kelas Kata EDICT

Penanda	Arti
abbr	singkatan
adj	adjektiva-i (<i>keiyoushi</i>)
adv	adverbia (<i>fukushi</i>)
adv-n	kata benda adverbial
adj-na	adjektiva-na (<i>keiyoudoushi</i>)
adj-no	kata benda yang bisa diberi partikel genitif “no”
adj-pn	adjektiva pra-kata benda (<i>rentaishi</i>)
adj-s	adjektiva khusus (misal <i>ookii</i>)
adj-t	adjektiva “taru”
arch	kuno
ateji	bacaan <i>ateji</i> dari kanji
aux	frasa atau kata pembantu
aux-v	verba pembantu
conj	konjungsi
col	bahasa percakapan
exp	ekspresi (frasa, klausa, dan sebagainya)
ek	eksklusif <i>kanji</i> , jarang ditulis dengan <i>kana</i>
fam	bahasa akrab
fem	bahasa atau istilah feminim
gikun	bacaan (arti) <i>gikun</i>
gram	istilah tata bahasa
hon	bahasa honorifik (<i>sonkeigo</i>)

hum	bahasa rendah diri (<i>kenjougo</i>)
id	idiom
int	interjeksi (<i>kandoushi</i>)
iK	penggunaan <i>kanji</i> yang tidak umum
ik	penggunaan <i>kana</i> yang tidak umum
io	penggunaan <i>okurigana</i> yang tidak umum
MA	istilah dunia bela diri
male	bahasa atau istilah maskulin
m-sl	<i>slang</i> komik Jepang
n	nomina umum (<i>futsuumeishi</i>)
n-adv	nomina adverbial (<i>fukushitekimeishi</i>)
n-t	nomina temporal (<i>jisoumeishi</i>)
n-suf	nomina, digunakan sebagai akhiran
n-pref	nomina, digunakan sebagai awalan
neg	negatif (pada kalimat negatif, atau dengan kata kerja negatif)
neg-v	kata kerja negatif
num	bilangan
obs	kuno
obsc	langka
oK	menggunakan <i>kanji</i> kuno
ok	menggunakan penulisan <i>kana</i> kuno
pol	bahasa sopan (<i>teineigo</i>)
pref	awalan
prt	partikel
qv	<i>quod vide</i> (lihat entri lain)
sl	<i>slang</i>
suf	akhiran
uK	umumnya ditulis menggunakan <i>kanji</i>
uk	umumnya ditulis menggunakan <i>kana</i>
v1	verba <i>ichidan</i> (verba -ru)
v5	verba <i>godan</i> (verba -u, belum diklasifikasi lebih lanjut)
v5u	verba <i>godan</i> berakhiran 'u'
v5u-s	verba <i>godan</i> berakhiran 'u', kelompok khusus

v5k	verba <i>godan</i> berakhiran 'ku'
v5g	verba <i>godan</i> berakhiran 'gu'
v5s	verba <i>godan</i> berakhiran 'su'
v5t	verba <i>godan</i> berakhiran 'tsu'
v5n	verba <i>godan</i> berakhiran 'nu'
v5b	verba <i>godan</i> berakhiran 'bu'
v5m	verba <i>godan</i> berakhiran 'mu'
v5r	verba <i>godan</i> berakhiran 'ru'
v5k-s	verba <i>godan</i> , kelompok khusus <i>iku/yuku</i>
v5aru	verba <i>godan</i> , kelompok khusus <i>-aru</i>
v5uru	verba <i>godan</i> , kelompok kuno <i>uru</i> (bentuk kuno <i>eru</i>)
vi	verba intransitif
vs	nomina yang bisa diakhiri verba <i>suru</i>
vs-i	verba <i>suru</i> , tak beraturan
vs-s	verba <i>suru</i> , kelompok khusus
vz	verba <i>zuru</i> , bentuk alternatif verba <i>-jiru</i>
vk	verba <i>kuru</i> , kelas khusus
vt	verba transitif
vulg	kata atau ungkapan vulgar
X	istilah kotor

Penanda tersebut dituliskan di dalam kurung pada terjemahan pertama, dengan format:

(p_utama_1, p_utama_2, ...) (p_tambahan_1) (p_tambahan_2) ...

Yang dianggap penanda utama adalah klasifikasi kelas kata pada tata bahasa Jepang (misalnya nomina), sedangkan yang dianggap sebagai penanda tambahan adalah informasi non-gramatikal misalnya informasi bahwa kata tersebut umumnya ditulis menggunakan kana (diberi penanda uk).

Contoh dari suatu entri EDICT adalah:

勉強 [ベンキョウ] / (n, vs) (1) study/ (2) diligence/ (3) discount/reduction/ (P) /

Analisisnya adalah:

- *Kanji* dari kata tersebut adalah 勉強
- *Kana* dari kata tersebut adalah ベンキョウ
- Kata tersebut diberi penanda kelas kata n dan vs, berarti kata tersebut merupakan nomina dan bisa diberi akhiran *suru*
- Terjemahan Inggrisnya adalah *study*, *dilligence*, dan sebagainya.
- Kata tersebut diberi penanda P, yang berarti bahwa kata tersebut termasuk populer

2.9 SQLite

SQLite adalah mesin SQL kecil yang kecepatannya tidak kalah dari DBMS lainnya (<http://sqlite.org>). Program dengan lisensi public domain ini sudah mencapai versi 3. Pada dasarnya, SQLite adalah pustaka bahasa C untuk dilink oleh program yang membutuhkannya (`sqlite3.dll` untuk Windows dan `sqlite3.so` untuk Linux). Tersedia juga binding untuk banyak bahasa lain (misalnya `System.Data.SQLite.dll` untuk .NET). Jadi dia bukanlah program server yang berjalan sendiri dan menunggu koneksi dari program lain sebagaimana program seperti MySQL dan Oracle.

Seluruh database (definisi, tabel, indeks, dan data) pada SQLite disimpan dalam satu berkas. Berkas ini cross platform.

2.10 Framework .NET, C#, dan ASP.NET

2.10.1 Framework .NET

Setelah DOS, *platform* pemrograman unggulan Microsoft adalah Win32. Yang dimaksud Win32 adalah segenap pustaka fungsi guna membangun aplikasi untuk sistem operasi Windows. Pengembangan *platform* ini dimulai pada era 80-an. Saat itu C adalah bahasa pilihan dan OOP belum banyak digunakan. Win32 pada akhirnya menjadi *platform* pilihan banyak pemrogram, menyebabkan Windows menjadi sistem operasi desktop yang paling populer hingga saat ini.

Jika Win32 merupakan cerminan teknologi pengembangan software tahun 80-an, Framework .NET adalah platform pemrograman yang merupakan perwujudan teknologi awal milenium kedua. Framework .NET diciptakan untuk dapat memecahkan masalah yang banyak dihadapi dunia pemrograman masa kini secara lebih efisien. Metodologi terbaru seperti pemrograman berorientasi objek dan konsep software sebagai komponen tertanam dengan kuat di Framework .NET.

Framework .NET terdiri dari Common Language Runtime (CLR) dan pustaka kelas .NET. CLR adalah bagian dari Framework .NET yang mengelola program-program .NET yang dijalankan. CLR inilah yang mengatur hal-hal seperti pengalokasian memori, pengecekan tipe data, dan keamanan. Pustaka kelas .NET dibuat untuk menyelesaikan berbagai masalah umum mulai dari kelas yang berisi fungsi-fungsi matematika, kelas-kelas yang berhubungan dengan keamanan, kelas-kelas untuk membuat program antarmuka grafis, sampai kelas-kelas untuk pemrograman jaringan (Tien, 2001).

Framework .NET sejak awal dirancang untuk mendukung berbagai bahasa pemrograman. Contoh-contoh bahasa yang bisa dipakai untuk pemrograman .NET adalah C#, Managed C++, Visual Basic .NET, Jscript .NET, Visual J++ .NET, Pascal .NET, IronPython, dan Nemerle. Pustaka kelas .NET yang dibuat dengan suatu bahasa pemrograman bisa digunakan secara langsung oleh bahasa pemrograman .NET lainnya.

2.10.2 C#

Microsoft membuat C# seiring dengan pembuatan Framework .NET. Arsitek utama dalam pembuatan C# adalah Anders Hejlsberg yang sebelumnya berperan dalam pembuatan Borland Delphi dan Turbo Pascal. C# menjanjikan produktifitas dan kemudahan yang ada di Visual Basic dengan kemampuan dan fleksibilitas yang ada di C/C++.

Menurut spesifikasi bahasanya (ECMA-334 Committee), “C# (dibaca C sharp) adalah bahasa pemrograman yang sederhana, modern, berorientasi objek, dan *type-safe*. Bahasa ini akan dengan mudah dimengerti pemrogram C maupun C++. C# menggabungkan produktifitas tinggi dari bahasa-bahasa *Rapid Application Development* dan keampuhan C++”. Untuk mencapai produktifitas tinggi ini konsep-konsep sulit C++ disederhanakan dan fitur-fitur baru ditambahkan. Hal ini mungkin terasa mirip dengan Java, karena itulah C# bisa dianggap sebagai sepupu Java.

Inilah contoh program *Hello World* menggunakan C#:

```
using System;
```

```
class Halo
{
    static void Main()
    {
        Console.WriteLine("Halo, dunia!");
    }
}
```

Gambar 2.6 *Hello World* di C#

2.10.3 ASP.NET

ASP.NET adalah model pengembangan web terintegrasi yang menyediakan servis-servis yang diperlukan untuk membangun aplikasi web dengan kode minimum. ASP.NET adalah bagian dari Framework .NET, dan saat melakukan pemrograman ASP.NET seluruh pustaka kelas Framework .NET bisa diakses. Bahasa yang bisa digunakan adalah bahasa apapun yang mendukung Framework .NET seperti C# dan Visual Basic .NET (Walther, 2006).

Inilah contoh aplikasi ASP.NET menggunakan bahasa C# yang mengembalikan halaman HTML yang berisi tanggal pengaksesan halaman tersebut:

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System" %>

<html>
  <head>
    <title>Halo</title>
  </head>
  <body>
    <p>Waktu akses:
      <% Response.Write(DateTime.Now); %>
    </p>
  </body>
</html>
```

Gambar 2.7 Contoh aplikasi ASP.NET

2.11 JavaScript dan AJAX

2.11.1 JavaScript

JavaScript adalah bahasa pemrograman yang secara umum digunakan untuk pemrograman sisi klien yang berjalan di *browser* web (Zakas, 2005). Fokus utamanya adalah membantu pemrogram berinteraksi dengan halaman web dan *window* dari *browser* itu sendiri. JavaScript sedikit banyak didasarkan pada Java, bahasa pemrograman berorientasi objek yang populer digunakan pada *applet*. Walaupun JavaScript memiliki sintaks dan metodologi pemrograman yang mirip Java, JavaScript bukanlah versi “ringan” Java, namun jenis bahasa sendiri.

Dengan JavaScript, bisa dihasilkan suatu halaman web yang dinamis. Caranya adalah dengan memodifikasi DOM (*Document Object Model*), yaitu representasi halaman web tersebut di memori (Resig, 2006). Suatu halaman web direpresentasikan di DOM sebagai pohon yang bisa dijelajah. Setiap *node* pada pohon DOM memiliki *pointer* ke ibunya, saudaranya, dan anak-anaknya.

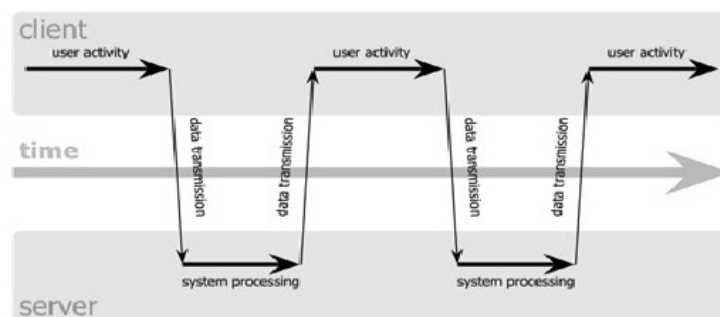
Modifikasi yang bisa dilakukan pada DOM misalnya adalah menghapus *node*, menambah *node*, dan mengubah suatu *node* (misal *style* CSS-nya). Dengan modifikasi tersebut, isi atau tampilan dari halaman web pun bisa dirubah secara programatis.

2.11.2 AJAX

AJAX adalah istilah yang dibuat oleh Jesse James Garrett dari Adaptive Path untuk menjelaskan komunikasi asinkronus klien-server yang dimungkinkan oleh objek XMLHttpRequest yang disediakan *browser* modern (Resig, 2006).

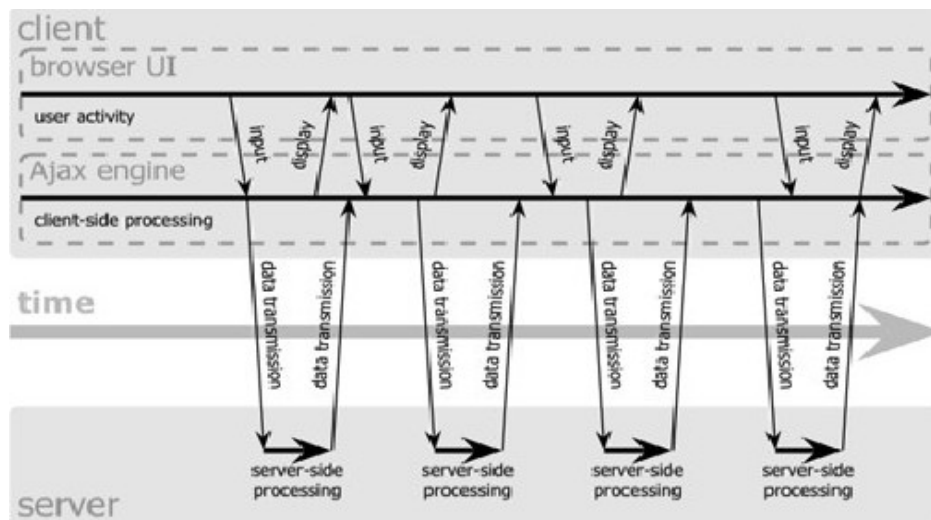
Singkatan dari *Asynchronous JavaScript and XML*, Ajax sebetulnya hanyalah istilah yang mencakupi teknik-teknik tertentu dalam pengembangan aplikasi web dinamis.

Pada aplikasi web klasik, untuk mendapatkan data baru dari server diperlukan *refresh* seluruh halaman. Ini menyebabkan penghamburan bandwidth karena pada umumnya banyak bagian dari halaman yang tidak berubah. Contohnya adalah *layout* yang mengelilingi isi utama halaman. Dengan model ini, pengguna juga harus menunggu relatif lama sebelum dapat melakukan aktivitas di halaman yang baru (Cho, 2006). Ini diilustrasikan pada Gambar 2.8 di bawah:



Gambar 2.8 Model aplikasi web klasik (Garrett)

Pada model AJAX, kode JavaScript bisa melakukan permintaan HTTP untuk mendapatkan data baru dan dinotifikasi saat datanya telah sampai. Dengan cara ini, yang perlu dikirim hanyalah data secukupnya. Setelah data sampai, DOM dapat dirubah sehingga isi halaman berubah tanpa perlu mengirim data redundan. Ini ditunjukkan pada Gambar 2.9 di bawah:



Gambar 2.9 Model aplikasi web menggunakan AJAX (Garrett)

AJAX tidak hanya mempercepat aplikasi web klasik, namun juga memungkinkan munculnya berbagai inovasi baru pada aplikasi web. Contohnya adalah *autocomplete* pada mesin pencarian (<http://suggest.google.com>) dan *instant messenger* berbasis web (<http://www.meebo.com>).

Namun AJAX bukanya tanpa kekurangan. Halaman yang dibuat secara dinamis tidak berkorespondensi dengan mesin *history browser*. Oleh karenanya, pengguna yang menekan tombol "*back*" belum tentu mendapatkan hasil yang diharapkan.

Selain itu, mesin pencari pada umumnya tidak menjalankan kode JavaScript yang dibutuhkan oleh fungsionalitas AJAX. Oleh karenanya, situs yang menginginkan isinya diindeks oleh mesin pencari, misalnya suatu situs *e-commerce*, lebih baik jangan membuat aplikasi yang sepenuhnya AJAX (www.interaktonline.com).

2.12 Pustaka Event Dean Edwards

Pustaka *event* Dean Edwards bisa diperoleh di <http://dean.edwards.name/weblog/2005/10/add-event/>. Di dalam pustaka tersebut tersedia fungsi-fungsi berkaitan dengan event yang bisa bekerja lintas *browser*.

Untuk menambah *event* ke ke elemen DOM tertentu, digunakan fungsi `addEventListener` berikut:

```
addEventListener(element, type, handler)
```

Contohnya adalah:

```
addEventListener(html.getElementById("foo"), "click",
    function(){alert('hello')});
```

Untuk menghapus *event* dari elemen DOM tertentu, digunakan fungsi `removeEvent`:

```
removeEvent(element, type, handler)
```

Contohnya adalah:

```
removeEvent(html.getElementById("foo"), "click", myHandler)
```

Pada fungsi yang menangani *event*, argumen pertama akan berisi informasi kontekstual tentang *event* yang terjadi. Pada argumen pertama tersebut juga terdapat fungsi `preventDefault()` yang mencegah *browser* melakukan aksi *default* untuk *event* tersebut dan `stopPropagation()` untuk mencegah propagasi *event* ke elemen DOM yang memuatnya. Di dalam fungsi yang menangani *event*, kata kunci `this` akan mengacu pada elemen DOM yang memunculkan *event* tersebut.

BAB III

ANALISIS, PERANCANGAN, DAN IMPLEMENTASI

3.1 Gambaran Umum Sistem

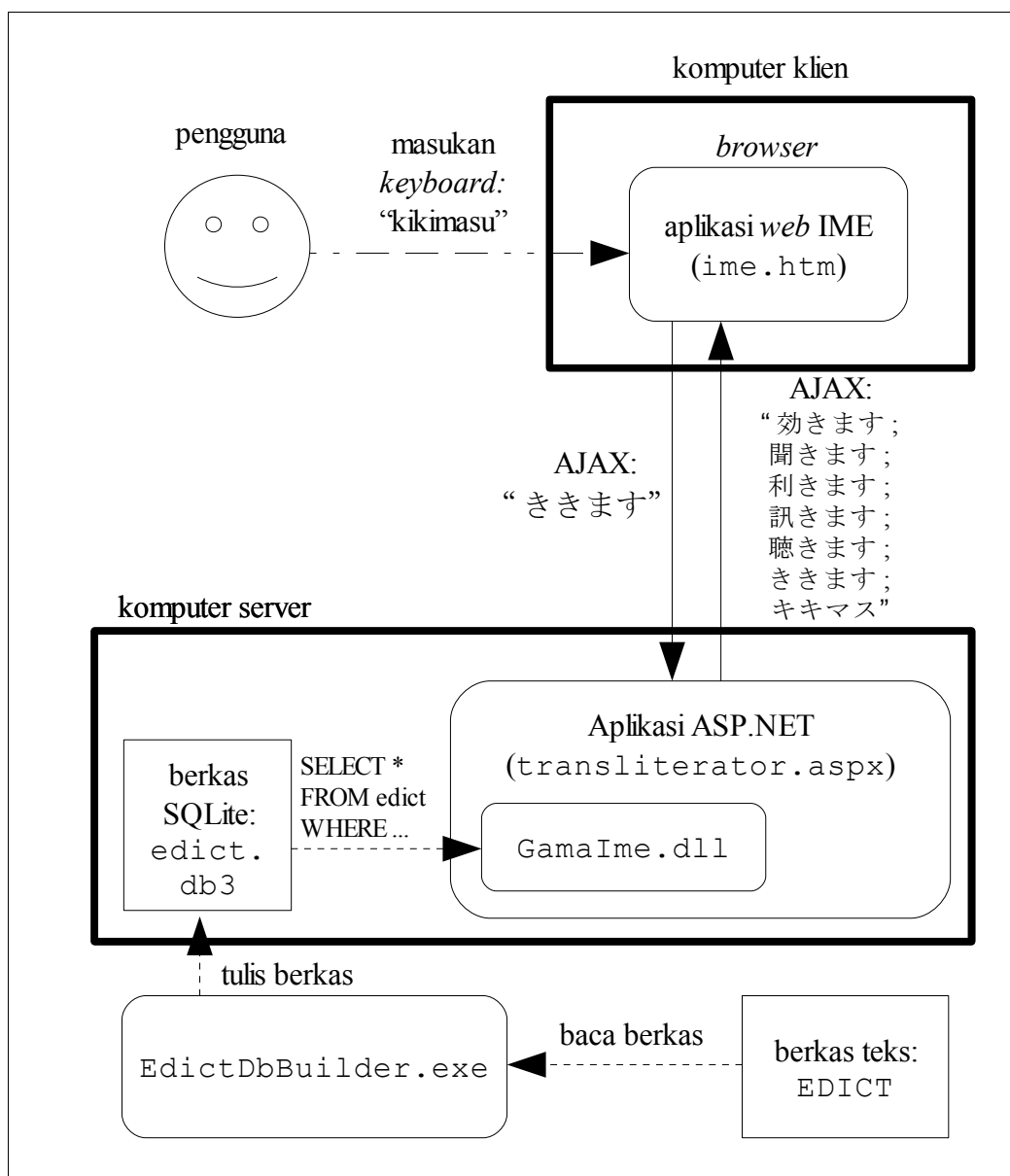
Di sisi pengguna yang terlihat adalah suatu aplikasi web. Pada aplikasi web tersebut terdapat suatu kotak teks tempat pengguna menuliskan masukan fonetis (misal “kikimasu”) dan menekan tombol spasi untuk mendapatkan transliterasinya.

Aplikasi web yang telah disebutkan akan melakukan panggilan AJAX saat pengguna menekan tombol untuk meminta transliterasi. Oleh karenanya, di sisi server terdapat aplikasi ASP.NET yang menerima panggilan AJAX tersebut. Lalu aplikasi ASP.NET itu akan mengoper masukan pengguna ke pustaka kelas utama yang akan mencari seluruh transliterasi yang mungkin. Setelah mendapatkan hasilnya, seluruh kemungkinan transliterasinya dikembalikan lagi ke aplikasi web agar dapat ditampilkan di *browser*. Sebagai contoh, dengan masukan pengguna “kikimasu” maka aplikasi ASP.NET akan mengembalikan string berikut ke *browser*, yang berupa kemungkinan transliterasi yang dipisahkan dengan titik koma: “効きます;聞きます;利きます;訊きます;聴きます;ききます;キキマス”

Sistem linguistik yang paling vital adalah pustaka kelas yang dinamakan *GamaIme.dll*. Di sinilah terdapat algoritma yang akan mencari transliterasi *kanji* dari masukan pengguna. Sebagai contoh, sistem ini bisa memberikan transliterasi dari “kikimasu” walaupun pada basis data hanya terdapat kata dasarnya, yaitu “kiku”.

Untuk keperluan transliterasi, dibutuhkan basis data kata dasar. Basis data ini akan dibaca oleh `GamaIme.dll`. Basis data yang berkasnya bernama `edict.db3` ini dihasilkan oleh program yang dinamakan `EdictDbBuilder` dari berkas teks kamus EDICT.

Hubungan seluruh komponen IME tersebut adalah sebagai berikut:



Gambar 3.1 Hubungan Komponen IME

Pada tulisan ini, akan dibahas seluruh komponennya mulai dari lapisan paling bawah yaitu `EdictDbBuilder`.

3.2 Komponen Pembangun Basis Data: `EdictDbBuilder.exe`

3.2.1 Tinjauan

Data kata dasar bersumber dari berkas teks EDICT versi 2005-12-08. Setelah `GamaIme.dll` menentukan kata dasar yang perlu dicari, ia akan mencarinya *kanji*-nya di data EDICT tersebut. Secara teknis, bisa saja berkas teks EDICT dibaca langsung untuk keperluan transliterasi. Namun proses *parsing* berkas teks membutuhkan waktu yang relatif lama, terlebih lagi jumlah kata pada EDICT berjumlah di atas 100 ribu. Alasan lain untuk tidak menggunakan berkas orisinal EDICT adalah karena terdapat informasi yang tidak diperlukan untuk transliterasi, seperti arti kata³. Oleh karenanya, dikembangkan suatu program C# bernama `EdictDbBuilder` yang akan mengubah berkas teks tersebut ke berkas basis data SQLite yang memungkinkan pencarian cepat dan hanya berisi data yang relevan.

3.2.2 Analisis Permasalahan

`EdictDbBuilder` pertama harus menyaring data-data yang relevan untuk transliterasi dari berkas teks EDICT. Data tersebut lalu harus disimpan

3 Arti kata pada EDICT juga secara umum tidak berguna untuk membantu memilih dua kanji yang memiliki perbedaan nuansa arti. Sebagai contoh, baik 直す dan 治す (keduanya *naosu*) sama-sama diberi arti “(1) to cure/to heal/(2) to fix/to correct/to repair” sehingga tidak berguna untuk mengetahui perbedaan penggunaan kedua kanji tersebut.

menjadi berkas basis data SQLite untuk digunakan dalam proses transliterasi.

3.2.3 Analisis Kebutuhan Sistem

EdictDbBuilder merupakan program C# 2.0 yang menggunakan pustaka `System.Data.SQLite` untuk Windows. Oleh karenanya untuk menjalankan program ini diperlukan Framework .NET 2.0 pada sistem operasi minimal Windows XP Service Pack 2. Program ini juga memerlukan berkas EDICT untuk dapat membangun berkas keluarannya.

3.2.4 Tabel edict

Hanya ada satu tabel pada basis data yaitu tabel `edict` yang memiliki *field-field* sebagai berikut:

Tabel 3.1 Field-Field Tabel edict

No.	Nama field	Tipe data	Keterangan
1	<u>kana</u> *	string	Cara membaca kata yang ditulis dalam <i>hiragana</i>
2	<u>kanji</u> *	string	Cara menulis kata dalam <i>kanji</i> , <i>hiragana</i> , atau <i>katakana</i>
3	tags	string	Kelas-kelas kata, dibatasi oleh pipa “ ”
4	pop	integer	Tingkat kepopuleran kata
5	uk	bool	Penanda apakah kata lebih umum ditulis menggunakan <i>hiragana</i>

Tidak semua data pada EDICT dimasukkan ke dalam tabel basis data. Yang dimasukkan hanyalah data yang relevan untuk keperluan transliterasi. Secara lebih spesifik, arti kata dan penanda-penanda yang tidak relevan akan

diabaikan. Penjelasan sepenuhnya mengenai data-data yang dibangun dan dimasukkan ada pada penjelasan untuk masing-masing *field*.

3.2.4.1 Field kana

Field string kana berisi cara membaca suatu kata. Setelah `GamaIme.dll` menentukan kata-kata dasar yang perlu dicari untuk keperluan transliterasi, *field* inilah yang akan di-*query* untuk mencari baris-baris tabel yang relevan. *Field* ini bisa saja ditulis dalam *hiragana*, *katakana*, maupun sistem *romaji* yang menyediakan pemetaan satu-satu ke *kana* seperti *Nihon-shiki*. *Hiragana* dipilih karena pada hampir seluruh entri EDICT yang memiliki *kanji*, cara membacanya ditulis dengan *hiragana*.

3.2.4.2 Field kanji

Field string kanji berisi cara menulis suatu kata. Jika kata tersebut hanya ditulis dalam *hiragana*, maka *field* kana dan kanji isinya sama. Jika suatu kata ditulis menggunakan *katakana*, maka *field* kana akan berisi cara membacanya dalam *hiragana*, sedangkan *field* kanji berisi penulisannya dalam *katakana*. Barulah jika suatu kata memiliki penulisan menggunakan *kanji*, *field* kanji benar-benar akan berisi *kanji*.

3.2.4.3 Contoh Pengisian (kana, kanji)

Inilah contoh semua kemungkinan yang berbeda dalam pengisian *field* kana dan kanji pada pembuatan basis data:

1. Kata yang tidak memiliki *kanji* dan umumnya ditulis menggunakan

hiragana. Pada EDICT, entrinya hanya berisi *hiragana*. Contohnya adalah “kichinto”:

きちんと / (adv,vs) precisely/accurately/ (P) /

Field kana dan kanji akan berisi sama:

- kana: きちんと
- kanji: きちんと

2. Kata yang tidak memiliki *kanji* dan umumnya ditulis menggunakan *katakana*. Pada EDICT, entrinya hanya berisi *katakana*. Contohnya adalah “terebi”:

テレビ / (n) television/TV/ (P) /

Field kana akan berisi penulisannya dalam *hiragana*:

- kana: てれび
- kanji: テレビ

3. Kata yang memiliki *kanji* yang pada EDICT cara membacanya ditulis menggunakan *hiragana* (kasus yang umum). Contohnya adalah “aruku”:

歩く [あるく] / (v5k) to walk/ (P) /

Hiragana-nya dan *kanji*-nya akan disalin masing-masing ke field kana dan kanji:

- kana: あるく
- kanji: 歩く

4. Kata yang memiliki *kanji* yang pada EDICT cara membacanya ditulis

menggunakan *katakana* atau gabungan *katakana-hiragana*⁴. Ini adalah kasus yang jarang. Terlebih lagi, pada beberapa kasus tersebut terdapat entri lain yang memiliki penulisannya dalam *hiragana* sepenuhnya. Contohnya adalah “momo”:

桃 [モモ] / (n) peach/prunus persica (tree) /
--

Dalam hal ini, penulisannya yang dalam *katakana* akan dirubah ke *hiragana* sebelum dimasukkan ke field *kana*:

- kana: もも
- kanji: 桃

3.2.4.4 (kana, kanji) sebagai Kunci Primer

Kunci primer pada tabel *edict* adalah (kana, kanji). Ini karena suatu bacaan bisa memiliki banyak kemungkinan *kanji* dan suatu *kanji* bisa memiliki lebih dari satu cara membaca. Yang unik seharusnya adalah gabungan bacaan dan *kanji* tertentu.

Namun pada kenyataannya, di berkas *EDICT* sendiri ini tidak berlaku karena dua kasus:

1. Suatu kata yang memiliki lebih dari satu arti, namun dipecah menjadi banyak entri. Contohnya adalah kata “menbou” yang memiliki 2 entri:

めん棒 [めんぼう] / (n) cotton swab/ めん棒 [めんぼう] / (n) rolling pin/
--

⁴ Cara membaca yang ditulis dalam *katakana-hiragana* muncul pada kata-kata yang cara menulisnya juga gabungan (misal *katakana-kanji*). Contohnya adalah “mekishikowan” メキシコ湾 yang cara membacanya ditulis sebagai メキシコわん.

Seharusnya, kedua entri ini digabung menjadi satu entri seperti:

めん棒 [めんぼう] / (n) (1) cotton swab / (2) rolling pin /

Ini mengikuti konvensi umum pada EDICT, seperti entri “shimobe” yang diasosiasikan dengan 2 arti:

僕 [しもべ] / (n) (1) manservant / (2) servant (of God) /

2. Suatu kata yang sebetulnya sama persis, namun memiliki dua entri karena salah satunya diberi bacaan dalam *hiragana*, dan yang lain diberi bacaan dalam *katakana*. Pada EDICT, gabungan (*kana*, *kanji*) tersebut unik, namun dalam proses pembuatan basis data tidaklah lagi unik karena bacaan *katakana* akan dirubah menjadi *hiragana*. Contohnya adalah “momo”:

桃 [もも] / (n) peach/prunus persica (tree) / (P) /
桃 [モモ] / (n) peach/prunus persica (tree) /

Karena dua masalah pada EDICT tersebut, pada `EdictDbBuilder` terdapat kode untuk menangani konflik kunci primer. Jika terjadi konflik saat berusaha memasukkan suatu entri, maka entri yang pertama kali dimasukkanlah yang akan dipakai sedangkan entri lainnya akan diabaikan.

3.2.4.5 Field tags, `edict_allowed_tags.txt`, dan `edict_tags_transform.txt`

Field string tags berisi kelas-kelas kata yang relevan untuk keperluan transliterasi. Sebagai contoh, penanda `v1` yang menandakan bahwa kata yang bersangkutan adalah verba *ichidan* (-ru) adalah penanda yang penting, sebab fakta bahwa suatu kata merupakan `v1` akan menentukan bagaimana bentuk infleksinya

yang mungkin. Di lain pihak, penanda MA yang menandakan bahwa kata yang bersangkutan merupakan istilah bela diri tidaklah relevan untuk keperluan transliterasi.

Dalam tahap transliterasi, GamaIme.dll akan menganalisis *field* tags ini untuk menentukan apakah kata yang bersangkutan bisa diinfleksi menurut aturan tata bahasa yang telah disimpulkan. Sebagai contoh, jika pengguna meminta transliterasi dari “kikanai” dan GamaIme.dll telah menyimpulkan bahwa:

“kikanai” bisa diperoleh dari kata dasar “kiku” yang tipenya “v5k”, dengan mengubah akhiran < “-ku” menjadi かふい “kanai”

Maka entri “kiku” berikut:

聞く [きく] / (v5k) (1) to hear/to listen/ (2) to ask/ (P) /
--

akan diinfleksi karena memiliki penanda v5k, sedang entri “kiku” berikut:

菊 [きく] / (n) chrysanthemum/ (P) /

akan diabaikan karena tidak memiliki penanda v5k.

Terdapat berkas `edict_allowed_tags.txt` yang berisi daftar pendanda-penanda yang dianggap relevan. Penanda di luar daftar tersebut tidak akan dimasukkan ke dalam *field* tags. `edict_allowed_tags.txt` merupakan berkas yang akan dijadikan *embedded resource* dalam program `EdictDbBuilder.exe`.

Format `edict_allowed_tags.txt` adalah tiap penanda yang dianggap relevan ditulis di tiap baris. Baris kosong akan diabaikan. Baris yang

diawali % akan dianggap *comment* dan juga diabaikan.

Inilah isi `edict_allowed_tags.txt` yang berisi daftar penanda yang dianggap relevan:

```
adj
adv
adj-na
adj-no
adj-pn
adj-t
aux
aux-v
conj
exp
int
n
n-adv
n-t
n-suf
n-pref
num
pref
prt
suf
v1
v5u
v5u-s
v5k
v5g
v5s
v5t
v5n
v5b
v5m
v5r
v5k-s
v5aru
v5uru
vi
vs
vs-i
vs-s
vz
vk

%not existing in documentation
adv-to
```

Gambar 3.2 `edict_allowed_tags.txt`

Pada *field* tags sendiri, tiap penanda akan dibatasi dengan pagar. Sebagai contoh, untuk entri “benkyou” berikut:

勉強 [べんきょう] / (n,vs) (1) study/(2) diligence/(3) discount/reduction/(P) /
--

isi dari *field* tags adalah “|n|vs|”. “1”, “2”, “3”, dan “P” tidak dimasukkan *field* tags karena tidak didefinisikan dalam `edict_allowed_tags.txt`.

Alasan untuk tidak menjadikan tiap penanda tersebut sebagai *field boolean* yang berdiri sendiri di tabel `edict` adalah jumlahnya yang sangat banyak. Pada SQLite, tiap *field* secara internal direpresentasikan sebagai *string*. Jadi dengan asumsi bahwa terdapat 40 penanda⁵, dan karena *field boolean* di SQLite pasti⁶ memakan tempat 2 *byte*, maka untuk tiap entri diperlukan tempat 80 *byte*. Ini bisa dibandingkan dengan 7 *byte* untuk menyimpan *string* “|n|vs|” pada contoh. Alasan lain adalah karena pengelompokan logis, sebab dalam salah satu tahap transliterasi pada `GamaIme.dll`, terdapat tahap pencocokan kelas kata seperti telah dijelaskan di awal bagian ini. Pencocokan dilakukan dengan mencari *substring* yang bersesuaian pada *field* tags.

Terdapat juga berkas `edict_tags_transform.txt`. Ini berguna untuk mengubah suatu penanda pada EDICT menjadi penanda lain yang diinginkan. Berkas ini juga akan dijadikan *embedded resource* dalam program `EdictDbBuilder.exe`.

Format tiap barisnya adalah:

⁵ Pada `edict_allowed_tags.txt` yang digunakan, terdapat 41.

⁶ Jika nilainya berupa *boolean* '1' atau '0', sebab SQLite tidak akan menolak jika diberi *string* lain.

```
[penanda_awal][TAB][penanda_baru]
```

Baris kosong maupun baris yang diawali penanda *comment* % akan diabaikan.

Fasilitas untuk mengubah penanda dibuat untuk mengakomodasi entri pada EDICT dengan penanda $v5r-i$ (verba -u baris r , tak beraturan). Kata dengan penanda tersebut hanyalah “aru” (在る atau 有る) yang bentuk negatifnya merupakan kata dengan *kanji* lain yaitu “nai” (無い). Untuk semua infleksi lainnya, “aru” berperilaku sama dengan verba -u baris r yang beraturan, yang pada EDICT diberi penanda $v5r$. Oleh karenanya, akan jauh lebih menyederhanakan aturan transliterasi jika $v5r-i$ dianggap sama dengan $v5r$ ⁷.

Inilah satu-satunya baris pada `edict_tags_transform.txt`:

```
v5r-i      v5r
```

Gambar 3.3 `edict_tags_transform.txt`

Fungsi lain transformasi penanda adalah memberikan penyamaan sementara untuk penanda-penanda khusus jika aturan tata bahasanya belum diketahui. Sebagai contoh, jika kita belum menemukan aturan tata bahasa untuk $v5u-s$ ($v5u$ tak beraturan), kita bisa menyamakannya untuk sementara dengan $v5u$.

3.2.4.6 Field pop

⁷ Satu-satunya kekurangan pada pendekatan ini adalah diterimanya bentuk negatif yang secara gramatikal tidak ada yaitu “aranai”. Bentuk ini berlaku pada $v5r$ tapi tidak untuk $v5r-i$. Namun sebagai perbandingan, bentuk salah ini juga diterima oleh IME Windows XP. Sebagai *false positive*, ini tidak bermasalah sebab pada bahasa Jepang memang tidak ada kata “aranai”.

Field integer ini menunjukkan tingkat kepopuleran kata, dengan 0 sebagai nilai *default* suatu kata. Nilai tersebut tidak langsung tertulis pada berkas teks EDICT, namun dihitung dengan cara yang dijelaskan di bagian ini.

Saat `GamaIme.dll` melakukan *query*, ada kemungkinan bahwa baris yang dikembalikan lebih dari satu. Sebagai contoh, saat pengguna meminta transliterasi dari “miru”, ada 5 entri EDICT yang bersesuaian:

```
海松 [みる] /(n) a type of seaweed/
看る [みる] /(oK) (v1) to look after/to take care of/(P)/
観る [みる] /(oK) (v1) to view (i.e. flowers, movie)/
見る [みる] /(v1) (1) to see/to watch/(2) (as an auxiliary
  verb) to try/(P)/
診る [みる] /(v1) to examine (medical)/(P)/
```

Idealnya, transliterasi *default* yang diberikan adalah yang diinginkan pengguna. Dengan adanya nilai kepopuleran, seluruh transliterasi yang diperoleh akan diurutkan berdasarkan kepopuleran⁸ sehingga memperbesar kemungkinan pengguna langsung memperoleh transliterasi yang diinginkan.

Terdapat berkas `edict_popularity_tags.txt` untuk mendefinisikan nilai kepopuleran suatu penanda, yang bisa positif maupun negatif. Nilai awal kepopuleran adalah 0, dan untuk tiap penanda pada entri yang diberi nilai kepopuleran, nilainya akan ditambahkan. Berkas `edict_popularity_tags.txt` dijadikan *embedded resource* dalam program `EdictDbBuilder.exe`.

Format tiap barisnya adalah:

```
[penanda] [TAB] [nilai]
```

⁸ Perkecualiannya adalah kalau salah satu entri memiliki penanda `uk`, yang akan dijelaskan di bagian berikutnya.

Baris kosong maupun baris yang diawali penanda *comment* % akan diabaikan.

Penanda P yang berarti “populer” diberi nilai 2. Selain itu, penanda yang berkonotasi bahwa kata tersebut tidak umum seperti arch (berarti kata kuno) diberi nilai -1. Inilah lengkapnya:

```
% Every EDICT item is considered to have popularity 0
(neutral)
% Tags defined in this file will modify the popularity
% There can be many of these tags per item, so it is
additive

% The tags themselves will "dissapear",
% i.e., is not represented in the item's tag list

% Format:
% [tag][tab][value]

P      2
arch  -1
iK    -1
ik    -1
io    -1
obs   -1
obsc  -1
oK    -1
ok    -1
```

Gambar 3.4 edict_popularity_tags.txt

Sebagai contoh, entri “miru” berikut:

```
見る [みる] /(v1) (1) to see/to watch/(2) (as an auxiliary
verb) to try/(P)/
```

akan diberi nilai kepopuleran 2 karena memiliki penanda P. Penanda v1, 1, dan 2 tidak mempengaruhi kepopuleran karena tidak didefinisikan di dalam edict_popularity_tags.txt. Sedangkan entri “uta” berikut:

```
唄 [うた] /(oK) (n,n-suf) song/(P)/
```

akan diberi kepopuleran 1 karena memiliki penanda P (+2) dan oK (-1). Penanda n maupun n-suf tidak mempengaruhi kepopuleran karena tidak didefinisikan di dalam `edict_popularity_tags.txt`.

3.2.4.7 Field uk

Field boolean ini menandakan ada atau tidaknya penanda uk (*usually kana*) pada entri EDICT. Keberadaan penanda uk berarti bahwa kata tersebut lebih umum ditulis menggunakan *hiragana* walaupun memiliki *kanji*.

Contoh entri EDICT yang memiliki penanda uk adalah “gomi”:

塵 [ごみ] / (n) (uk) rubbish/trash/garbage/refuse/ (P) /

Umumnya, `GamaIme.dll` akan memberikan transliterasi *kanji*-nya terlebih dahulu, lalu setelahnya transliterasinya dalam *hiragana* dan *katakana*. Namun jika *field uk* bernilai *true*, `GamaIme.dll` akan meletakkan transliterasi *hiragana*-nya mendahului *kanji*-nya.

Field ini dipisahkan dari *field tags* sebab pada proses transliterasi, keduanya digunakan di tahap yang berbeda. *Field tags* digunakan untuk menyeleksi apakah baris tersebut memenuhi syarat kelas kata yang ingin ditransliterasi, sedang *field uk* digunakan untuk keperluan pengurutan hasil transliterasi.

3.2.5 Implementasi

Program merupakan program *command-line* yang dibuat menggunakan

bahasa C#. Selain berkas-berkas teks `edict_allowed_tags.txt`, `edict_tags_transform.txt`, dan `edict_popularity_tags.txt` yang di-*embed* ke dalam program, terdapat berkas kode sumber `Edict.cs` dan `BuildDb.cs`.

`Edict.cs` berisi kelas `EdictEntry` yang akan melakukan parsing pada suatu baris EDICT untuk mendapatkan data-data yang siap dimasukkan pada basis data.

`BuildDb.cs` berisi metode `Main` pada kelas `BuildDb` yang akan membaca tiap baris berkas EDICT dan membangun basis data SQLite-nya.

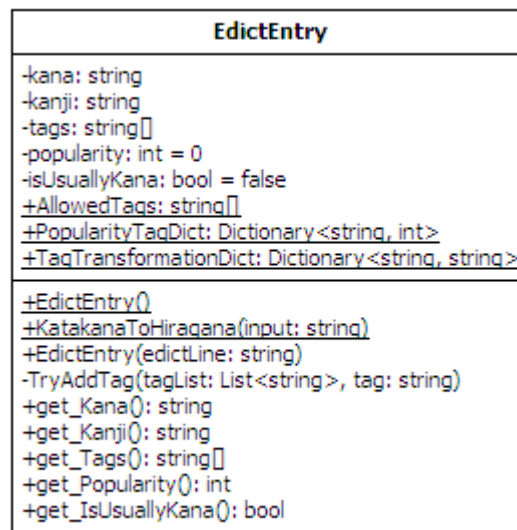
3.2.6 Kelas EdictEntry

3.2.6.1 Tinjauan

Kelas `EdictEntry` merepresentasikan suatu entri EDICT yang data-datanya telah diadaptasi untuk disimpan di basis data. Tugas utama kelas ini adalah melakukan parsing pada suatu baris entri EDICT sehingga bisa diakses dengan mudah sebagai suatu objek C#.

3.2.6.2 Perancangan Kelas

Inilah diagram kelas `EdictEntry`:



Gambar 3.5 Diagram Kelas EdictEntry

Atribut statik AllowedTags, TagTransformationDict, dan PopularityTagDict masing-masing dibaca dari berkas-berkas teks edict_allowed_tags.txt, edict_tags_transform.txt, dan edict_popularity_tags.txt yang di-embed ke dalam program. Kode untuk parsing berkas-berkas tersebut ada di konstruktor statik. Diagram yang menunjukkan hubungan kelas ini dengan kelas-kelas lain pada program ada di lampiran A.

3.2.6.3 Metode Statik string EdictEntry.KatakanaToHiragana(string input)

Karena bacaan pada basis data akan disimpan dalam *hiragana*, sedangkan terdapat entri *katakana* pada EDICT, maka diperlukan fungsi untuk mengubah *katakana* ke *hiragana*. Inilah tugas metode statik KatakanaToHiragana:

```
public static string KatakanaToHiragana(string input)
{
```

```

StringBuilder sb = new StringBuilder(input.Length);
foreach(char c in input)
{
    if(0x30a1 <= c && c <= 0x30f4)
    {
        sb.Append((char)(c - 0x60));
    }
    else
    {
        sb.Append(c);
    }
}
return sb.ToString();
}

```

Gambar 3.6 Metode KatakanaToHiragana

Karena titik kode tiap karakter *hiragana* pada Unicode berbeda sejumlah konstan dengan karakter *katakana* yang bersesuaian, maka algoritma perubahan tiap karakternya sangatlah sederhana:

1. Cek apakah karakter tersebut merupakan *katakana*, yang titik kodenya dimulai dari U+30A1 sampai U+30F4. Sebetulnya pada Unicode *katakana* didefinisikan sampai U+30FE namun karakter-karakter tersebut diabaikan karena:
 - a) Merupakan karakter yang tidak digunakan pada penulisan suara kata EDICT, seperti katakana “ke” kecil ケ.
 - b) Merupakan simbol yang tidak memiliki padanan di *hiragana*, misalnya simbol pemanjang suara ー.
2. Jika masuk jangkauan tersebut, kurangi nilai numeralnya sebanyak 0x60 untuk mendapatkan *hiragana* yang bersesuaian. Sebagai contoh, ア (katakana “a”, U+30A2) akan menjadi あ (hiragana “a”, U+3042). Jika tidak, karakternya tidak usah diubah. Sebagai contoh karakter pemanjang

suara — tidak akan diubah.

3.2.6.4 Metode void `EdictEntry.TryAddTag(List<string> tagList, string tag)`

Metode `TryAddTag` menerima argumen daftar penanda yang sudah dimasukkan `tagList` dan penanda baru `tag` yang ingin dicoba dimasukkan.

Kemungkinannya adalah:

1. Jika `tag` termasuk di daftar penanda pada `edict_allowed_tags.txt` (datanya sudah dimuat di atribut statik `AllowedTags`), maka akan ditambahkan ke `tagList`.
2. Jika `tag` adalah “uk”, maka atribut `isUsuallyKana` akan diset menjadi `true`.
3. Jika `tag` didefinisikan di `edict_popularity_tags.txt` (datanya sudah dimuat di atribut statik `PopularityTagDict`), maka nilai atribut `popularity` akan diubah.
4. Jika `tag` didefinisikan di `edict_tags_transform.txt` (datanya sudah dimuat di atribut statik `PopularityTagDict`), maka metode ini akan dipanggil lagi secara rekursif dengan argumen barunya merupakan penanda hasil transformasi.
5. Jika tidak, maka tidak terjadi perubahan apa-apa. Dengan kata lain, penanda tersebut diabaikan.

3.2.6.5 Konstruktor `EdictEntry()`

Parsing suatu baris EDICT dilakukan di konstruktor `EdictEntry` yang

menerima argumen *string* `edictLine`. Pertama dipisahkan antara “arti” (tempat penanda juga berada) dengan katanya (bacaan dan penulisan). Pemisahannya adalah karakter “/”:

```
int slashIndex = edictLine.IndexOf('/');
string translation = edictLine.Substring(slashIndex);
```

Gambar 3.7 Pemisahan Arti pada Konstruktor `EdictEntry`

Lalu atribut *kana* dan *kanji* diisi berdasarkan *string* di sebelah kiri “/”.

Kemungkinannya ada 2 yaitu:

- Tidak ada *kanji*, berarti hanya terdapat *hiragana* atau *katakana* yang diikuti spasi. Contohnya adalah “amerika” yang ditulis dalam *katakana*:

“アメリカ ”

dan “potsunto” (sendiri) yang ditulis dalam *hiragana*:

“ぽつんと ”

- Ada *kanji*, berarti pertama *kanji*-nya (termasuk *okurigana*) ditulis, diikuti spasi, lalu “[“, lalu bacaannya dalam *kana*, lalu “[”], dan terakhir spasi. Contohnya adalah “hitotsu” (satu):

“一つ [ひとつ] ”

Kode berikutnya menentukan mana kasus yang terjadi berdasarkan keberadaan “[“, dan selanjutnya mengisi atribut *kanji* dan *kana*:

```
int bracketIndex = edictLine.IndexOf('[');
if (bracketIndex == -1)
{
    this.kanji = edictLine.Substring(0, slashIndex - 1);
    this.kana = KatakanaToHiragana(this.kanji);
}
else
{
    this.kanji = edictLine.Substring(0, bracketIndex - 1);
```

```

    this.kana =
    KatakanaToHiragana(edictLine.Substring(bracketIndex + 1,
    slashIndex - bracketIndex - 3));
}

```

Gambar 3.8 Pengisian Bacaan dan Penulisan pada Konstruktor EdictEntry

Selanjutnya penanda yang ada dicari di dalam *string* translation yang telah diperoleh sebelumnya. Kemungkinannya ada dua:

- Di dalam kurung terdapat satu penanda. Contohnya adalah penanda *n* dan *P* pada string translation dari entri “hitotsu” berikut:

“/(*n*) one/(*P*)/”

- Di dalam kurung terdapat banyak penanda yang dipisahkan oleh koma. Contohnya adalah penanda *n* dan *vs* pada string translation dari entri “benkyou” (belajar):

“[べんきょう] /(*n,vs*) (1) study/(2) diligence/(3) discount/reduction/(*P*)/”

(pada contoh tersebut, terdapat juga penanda 1, 2, 3, dan *P* yang merupakan kasus pertama)

Kode berikutnya mencari penanda dengan mengantisipasi kedua kemungkinan tersebut, memindai dari kiri ke kanan dan melakukan operasi *substring* untuk mendapatkan penandanya. Jika penanda ditemukan, maka metode `TryAddTag` dipanggil. Inilah kodenya:

```

List<string> tagList = new List<string>();
int startIndex = -1;
bool inParanthesis = false;
for(int i = 0; i < translation.Length; i++)
{
    char c = translation[i];
    if(c == '(')

```

```

    {
        inParanthesis = true;
        startIndex = i + 1;
    }
    else if(c == ',' && inParanthesis)
    {
        string tag = translation.Substring(startIndex, i
- startIndex);
        this.TryAddTag(tagList, tag);
        startIndex = i + 1;
    }
    else if(c == ')')
    {
        string tag = translation.Substring(startIndex, i
- startIndex);
        this.TryAddTag(tagList, tag);
        inParanthesis = false;
    }
}
this.tags = tagList.ToArray();

```

Gambar 3.9 Pengisian Penanda pada Konstruktor EdictEntry

Jika konstruktor telah selesai, berarti string yang diberikan telah di-*parse* dan data-datanya bisa diambil melalui berbagai properti yang ada.

3.2.7 Kelas BuildDb

3.2.7.1 Tinjauan

Pada kelas BuildDb terdapat pintu masuk program yaitu metode statik Main. Secara default, program akan membaca berkas teks EDICT “edict” dan mengeluarkan berkas SQLite “edict.db3”. Ini bisa diubah dengan memberikan argumen saat memanggil program. Sintaks programnya adalah:

```
EdictDbBuilder.exe [edictFile] [outputFile]
```

3.2.7.2 Perancangan Kelas

Inilah diagram kelasnya:

BuildDb
<code>-edictFile: string = "edict"</code> <code>-outputDbFile: string = "edict.db3"</code>
<code>-Main(args: string[]): int</code> <code>-Help()</code> <code>-PopulateDb(conn: SQLiteConnection, entries: List<EdictEntry>)</code> <code>-JoinTags(tags: string[]): string</code>

Gambar 3.10 Diagram Kelas BuildDb

3.2.7.3 Metode Statik `string BuildDb.JoinTags(string[] tags)`

Metode ini menerima larik *string* dan akan menggabungkannya menjadi bentuk seperti “|str_1|str_2|str_n|” yang sesuai dengan format *field* tags pada tabel `edict` yang akan dibuat.

3.2.7.4 Metode Statik `void BuildDb.Help()`

Metode ini menuliskan sintaks program pada terminal, dan akan dipanggil saat program diberi argumen gaya Unix “-h” atau “--help” maupun gaya DOS “/h”.

3.2.7.5 Metode Statik `void BuildDb.PopulateDb(SQLiteConnection conn, List<EdictEntry> entries)`

Metode ini menerima koneksi SQLite yang telah dispesifikasikan nama berkasnya dan *List EdictEntry* untuk dimasukkan datanya ke basis data. Dalam memasukkan data ke tabel, konversi yang dilakukan hanyalah `IsUsuallyKana` yang diubah dari `true/false` ke `1/0` dan `Tags` yang diubah

dari larik menjadi *string* dengan metode `JoinTags`.

Karena secara default SQLite melakukan penulisan ke berkas setiap operasi, maka memasukkan data secara mentah-mentah akan membutuhkan waktu yang sangat lama. Oleh karenanya, digunakan perintah SQL `BEGIN` dan `COMMIT` sehingga basis data hanya akan ditulis dari memori ke berkas setelah `COMMIT` dilakukan:

```
cmd.CommandText = "BEGIN;";
cmd.ExecuteNonQuery();

// kode INSERT, diabaikan di listing ini

cmd.CommandText = "COMMIT;";
cmd.ExecuteNonQuery();
```

Gambar 3.11 Penggunaan `BEGIN` dan `COMMIT` pada `PopulateDb`

Karena proses memasukkan data memakan waktu yang cukup lama, metode ini melaporkan prosesnya ke *console* setiap memasukkan 500 entri. Setelah selesai memasukkan seluruh entri, akan dilaporkan semua masalah yang ditemui (jika ada) dan jumlah entri yang berhasil dimasukkan.

3.2.7.6 Metode Statik `int BuildDb.Main(string[] args)`

Metode ini pertama menganalisis argumennya untuk menampilkan *help* atau mengubah nama berkas *default*. Lalu metode ini membuka berkas `EDICT` dan mengubah setiap barisnya menjadi objek `EdictEntry`. Setelahnya metode ini berusaha membuat berkas basis data dan membuat koneksinya. Terakhir, koneksi dan *List EdictEntry* diberikan ke metode `PopulateDb`.

3.3 Komponen Pentransliterasi Kanji: GamaIme.dll

3.3.1 Tinjauan

GamaIme.dll merupakan pustaka .NET yang dibuat dengan bahasa C#. Di sinilah terdapat mesin yang melakukan transliterasi *kanji*.

3.3.2 Analisis Permasalahan

GamaIme.dll harus bisa mentransliterasi dari *romaji* atau *hiragana* ke *kanji*. Data yang digunakan adalah basis data SQLite yang telah dibangun sebelumnya dengan program EdictDbBuilder.exe. Secara lebih spesifik, masukan yang diterima adalah:

- *string romaji*

Dan keluarannya adalah:

- Larik *string* pilihan transliterasi

Metode yang melakukan transliterasi adalah `Transliterate` dari kelas `KanjiTransliterator`. Format *string romaji* yang digunakan ada di 3.3.4.3. Untuk mengubah *string hiragana* ke *string romaji*, tersedia metode `Transliterate` dari kelas `HiraganaToRomajiTransliterator` untuk keperluan tersebut.

3.3.3 Analisis Kebutuhan Sistem

GamaIme.dll dikembangkan menggunakan bahasa C# 2.0 dan memanfaatkan pustaka kelas `System.Data.SQLite.dll` untuk Windows. Oleh karenanya untuk menggunakan pustaka kelas tersebut diperlukan

Framework .NET 2.0 yang berjalan di sistem operasi minimal Windows XP Service Pack 2.

3.3.4 Mesin Transliterasi Kanji

3.3.4.1 Tinjauan

Pada sisi klien, pengguna memberikan masukan berupa *romaji*. Aplikasi web sisi klien akan mengubahnya menjadi *kana* dan mengirimnya ke server. `GamaIme.dll` di sisi server berfungsi mencari seluruh transliterasinya yang mungkin. Secara lebih spesifik, fungsionalitas ini disediakan oleh metode `Transliterate` pada objek `KanjiTransliterator`.

Initinya, ada empat hal yang perlu dilakukan. Pertama adalah mencari seluruh kata dasar yang mungkin. Ini dilakukan dengan breadth-first search dan algoritma perantaraan aturan tata bahasa. Setelah daftar kata dasarnya diketahui, kedua adalah mencari penulisan kata dasar-kata dasar tersebut di basis data. Ketiga adalah menginfleksi kata-kata yang ditemukan pada basis data menjadi bentuk yang dimasukkan oleh pengguna. Terakhir adalah mensortir seluruh kata terinfleksinya agar hasil yang lebih relevan diletakkan di atas.

Untuk memahami proses pencarian kata dasar, perlu dipahami sebelumnya dua buah struktur data yang digunakan yaitu `Rule` dan `RuleInstance`. Tanpa dibahas terlebih dahulu mengenai bagaimana atributnya diorganisir secara internal, akan dijelaskan informasi yang terkandung di dalamnya.

`Rule` menspesifikasikan suatu aturan tata bahasa. Contohnya suatu objek `Rule` dapat menyimpan informasi berikut:

Suatu “v1” (verba ichidan) dapat diubah menjadi “neg” (bentuk negatif) dengan mengganti akhiran “ru” menjadi “nai”

Gambar 3.12 Contoh Informasi pada suatu Rule

Di lain pihak, `RuleInstance` berisi infleksi yang sudah diaplikasikan terhadap suatu kata tertentu. Contohnya suatu objek `RuleInstance` dapat menyimpan informasi berikut:

Jika terdapat suatu “v1” (verba ichidan) “taberu”, maka terdapat suatu infleksi yang mengganti akhiran “ru” dengan “nai”

Gambar 3.13 Contoh Informasi pada suatu RuleInstance

Pada Gambar 3.13, `RuleInstance`-nya merupakan aplikasi aturan pada Gambar 3.12 terhadap kata “taberu”. Namun `RuleInstance` juga bisa berisi informasi yang hanya dapat diperoleh dengan merantai banyak Rule, seperti pada contoh berikut:

Jika terdapat suatu “v1” (verba ichidan) “taberu”, maka terdapat suatu infleksi yang mengganti akhiran “ru” dengan “teiru”

Gambar 3.14 Contoh Informasi pada suatu RuleInstance yang diperoleh melalui Perantaraan

Pada contoh Gambar 3.14 tersebut, sebetulnya tidak ada Rule yang secara langsung mengubah verba ichidan menjadi bentuk tersebut. Jadi suatu `RuleInstance` tidak harus berkorespondensi dengan satu Rule.

Kunci dari pencarian kata dasar adalah suatu Rule bisa digunakan untuk pengambilan kesimpulan mundur. Sebagai contoh, misal terdapat `RuleInstance` berikut:

Jika terdapat suatu “neg” (kata negatif) “tabenai”, maka terdapat suatu infleksi yang mengganti akhiran “i” dengan “kat'ta”

Gambar 3.15 Contoh Informasi pada suatu RuleInstance untuk Dirantaikan

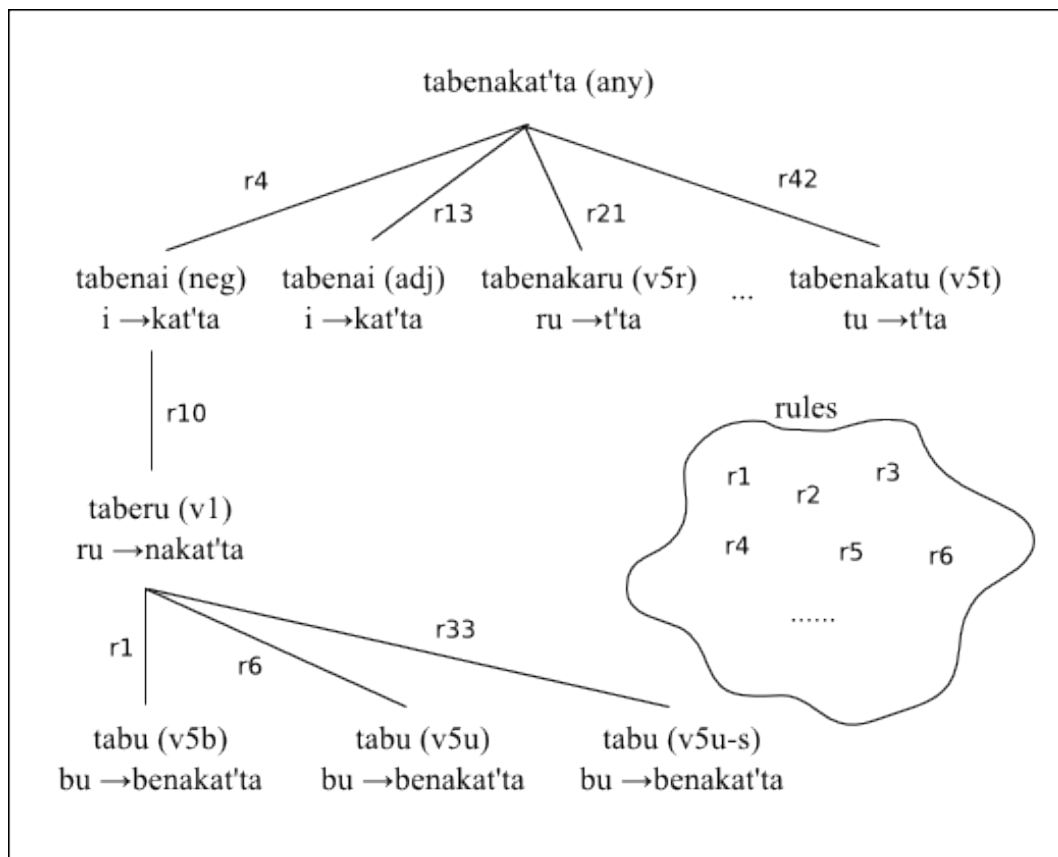
Dengan Rule pada Gambar 3.12, memungkinkan untuk disimpulkan bahwa “tabenai” bisa dibentuk dari “taberu”, sehingga bisa didapat RuleInstance berikut yang kata dasarnya berbeda dengan Gambar 3.15 namun hasil infleksinya sama:

Jika terdapat suatu “v1” (verba ichidan) “taberu”, maka terdapat suatu infleksi yang mengganti akhiran “ru” dengan “nakat'ta”

Gambar 3.16 Contoh Informasi pada suatu RuleInstance hasil Perantaian

Proses tersebut dinamakan perantaian.

Dengan perantaian dan *breadth-first search*, dimungkinkan dibentuk pohon pencarian kata dasar. Untuk ilustrasi pencarian kata dasarnya, digunakan contoh masukan “tabenakat'ta” (tidak makan, bentuk lampau). *Node* pertama hanyalah RuleInstance yang tidak melakukan perubahan apapun ke “tabenakat'ta”. Dari situ tiap *node* dicoba dirantaikan dengan seluruh aturan yang ada. Pohon pencariannya adalah sebagai berikut:



Gambar 3.17 Pohon Perantain “tabenakat'ta”

Dari seluruh `RuleInstance` yang ditemukan, akan dilakukan pencarian kata dasar ke basis data. Dari situ yang akan ditemukan hanyalah “taberu” (v1) dengan penulisan 食べる yang akan diinfleksi menjadi “tabenakat'ta” 食べなかつた.

Struktur data dan prosesnya akan dijelaskan secara lebih detil di bagian-bagian selanjutnya.

3.3.4.2 Penggunaan Romaji

Mesin transliterasi pada `GamaIme.dll` berbasis *romaji*. Secara lebih spesifik, *romaji* akan digunakan dalam perantain aturan tata bahasa (berikutnya

akan disingkat “perantaraan” saja). Alasannya adalah banyak aturan infleksi yang bisa dinyatakan secara lebih sederhana menggunakan *romaji*⁹.

Sebagai contoh, untuk mengubah suatu verba *godan* dasar menjadi bentuk potensial (misal “membaca” menjadi “bisa membaca”), aturannya menggunakan *romaji* bisa dinyatakan sebagai substitusi *string* sederhana:

ganti “u” pada akhir kata menjadi “eru”

Contohnya, “kiku” (menulis) menjadi “kikeru” (bisa menulis) dan “kau” (membeli) menjadi “kaeru” (bisa membeli).

Jika aturannya ditulis menggunakan *kana*, maka diperlukan klasifikasi lebih lanjut dari verba *godan*. Contohnya untuk baris ku:

ganti く “ku” pada akhir kata menjadi ける “keru”

Untuk baris u:

ganti う “u” pada akhir kata menjadi える “eru”

Dan semacamnya untuk kategori verba *godan* yang lain.

Dengan menggunakan *romaji*, jumlah aturan yang perlu ditulis bisa dikurangi.

3.3.4.3 Romanisasi yang Digunakan

Karena aturan tata bahasa Jepang berhubungan erat dengan fonologi bahasa Jepang, diperlukan sistem romanisasi yang merefleksikan struktur fonologis tersebut agar sistemnya sederhana.

Sistem romanisasi seperti Hepburn akan menimbulkan banyak masalah

⁹ Prasyaratnya adalah digunakan sistem romanisasi yang mempertahankan struktur fonologi bahasa Jepang. Lihat 3.3.4.3.

jika digunakan, karena struktur fonologis *kana* tidak dipertahankan. Sebagai contoh, し yang merupakan kana baris “s” dengan vokal “i” ditulis sebagai “shi” bukannya “si”. Contoh konkrit masalahnya akan diilustrasikan.

Terdapat aturan berikut dalam tata bahasa Jepang untuk mengubah bentuk dasar menjadi bentuk akar (Kim) atau konjungtif (Yoshida, 1996) yang berlaku pada verba *godan*:

ganti vokal “u” pada akhir kata menjadi vokal “i”

Dengan aturan tersebut dan pengetahuan akan fonologi bahasa Jepang, bisa disimpulkan bahwa はなし (“hanashi” di Hepburn dan “hanasi” di Nihon-shiki) berasal dari はなす “hanasu” dan たち (“tachi” di Hepburn dan “tati” di Nihon-shiki) berasal dari たつ (“tatsu” di Hepburn dan “tatu” di Nihon-shiki). Bila sistem Hepburn yang digunakan bersamaan dengan penggantian *string* sederhana, maka pada proses perantaraan “hanashi” akan dianggap berasal dari “hanashu” (はなしゅ) dan “tachi” akan dianggap berasal dari “tachu” (たちゅ). Sistem seperti Nihon-shiki tidak bermasalah pada kasus tersebut.

Namun karena sistem Hepburn lebih dikenal, pada mesin transliterasi diputuskan untuk menggunakan Nihon-shiki hanya pada kasus-kasus di mana sistem Hepburn bermasalah. Terdapat juga romanisasi yang tidak ditemukan di manapun untuk mengakomodasi penggunaan sistem hibrida ini dalam perantaraan. Untuk selanjutnya sistem hibrida ini akan disebut sebagai sistem romanisasi GamaIme.dll.

Inilah rangkuman dari sistem romanisasi tersebut:

- *Kana* baris a (あ), k (か), g (が), n (な), b (ば), p (ぱ), m (ま), y (や), dan

w (わ) sama baik di Hepburn maupun Nihon-shiki, dan GamaIme.dll tidak menggunakan sistem yang berbeda.

- Dalam transliterasi *kana* ke *romaji*, silabel “n” (ん) akan selalu ditulis sebagai n' (n diikuti apostrof). Contohnya もんげん akan ditransliterasi menjadi “mon'gen” walaupun sebetulnya “mongen” tidak akan menimbulkan ambiguitas tentang silabel “n”. Ini untuk mempermudah proses transliterasi *kana* ke *romaji*, sehingga tidak perlu tahu apakah karakter di depannya berpotensi menimbulkan ambiguitas.
- *Kana* baris s (さ) dan t (た) mengikuti Nihon-shiki karena konsistensi aturan fonologisnya diperlukan dalam perantaraan. Dengan ini, し menjadi “si”, ち menjadi “ti”, dan つ menjadi “tu”. *Kana* baris d (だ) juga menggunakan Nihon-shiki (ぢ menjadi “di”), karena jika menggunakan Hepburn maka ぢ akan menjadi “ji” yang sama dengan じ.
- *Kana* baris z (ざ) dan h (は) menggunakan Hepburn sehingga じ menjadi “ji” dan ふ menjadi “fu”. Ini tidak bermasalah karena tidak ada infleksi yang berhubungan dengan perubahan suara vokal pada baris tersebut.
- *youon* (misal しゃ) menggunakan Hepburn karena lebih umum (misal “cha” v.s. “tya”) dan hanya menimbulkan satu masalah yang bisa diatasi dengan mudah, seperti dijelaskan di bawah.
- つ (*sokuon*) ditransliterasi sebagai t'. Ini karena saat pengetesan ditemukan masalah pada infleksi bentuk ～ては (“teha” pada GamaIme.dll) menjadi bentuk informalnya ～ちゃ¹⁰ (“cha” pada GamaIme.dll) jika

10 Contohnya adalah 笑っちゃだめ (tidak boleh tertawa).

menggunakan Nihon-shiki. Menggunakan Nihon-shiki, “waraccha” akan dianggap berasal dari “waracteha” yang tidak bisa ditransliterasi ke *kana* menggunakan sistem yang normal. Dengan menggunakan t', penulisan *sokuon* tidak tergantung karakter di belakangnya sehingga “warat'cha” akan dianggap berasal dari “warat'teha” dan bisa ditransliterasi ke *kana*.

Untuk selanjutnya, *romaji* akan ditulis menggunakan transliterasi GamaIme.dll pada pembahasan yang spesifik dengan perantaraan.

3.3.4.4 Representasi Aturan

Aturan infleksi tata bahasa disimpan dalam sebuah berkas teks, yang pada saat runtime akan dimuat menjadi objek-objek Rule. Aturan harus mampu mengakomodasi keunikan infleksi yang mungkin pada bahasa Jepang, yaitu:

- a) Perubahan di akhir. Contohnya adalah 食べる “taberu” menjadi 食べた “tabeta”. Pada contoh tersebut “ru” berubah menjadi “ta”. Penambahan di akhir seperti 魚 “sakana” menjadi 魚だ “sakana da” juga dianggap perubahan, dengan yang berubah adalah *string* kosong “”.
- b) Infleksi yang memiliki *kanji*. Contohnya adalah 持つ “motu” menjadi 持っていく ”mot'teiku” (*hiragana*) yang bisa juga ditulis dengan *kanji* 持って行く.
- c) Penambahan di depan. Contohnya adalah 冒険 “bouken” menjadi 大冒険 “daibouken”.
- d) Perubahan suara pada *kanji* kata dasar. Contohnya adalah 来る “kuru” menjadi 来た “kita”.

Format aturannya adalah:

[penanda_sumber]/[pola_sumber]/[pola_tujuan]/[penanda_tujuan]

penanda_sumber menunjukkan untuk kelas kata apa saja aturan tersebut berlaku. Kelas katanya adalah kelas kata EDICT, kelas kata perantara yang dihasilkan dari penanda_tujuan (misal te), atau any. any berarti aturan tersebut berlaku untuk semua jenis kata. Formatnya adalah:

penanda_1{,penanda_2,penanda_3,...,penanda_n}

Sebagai contoh, untuk merubah *nomina* (misal “aidoru”) dan *keiyoudoshi* (misal “suki”) menjadi bentuk negatifnya, ditambahkan “janai” di belakang kata tersebut. Berarti aturan tersebut berlaku untuk kelas kata EDICT n dan adj-na. Maka penanda_sumber adalah:

n, adj-na

Sebagai contoh lain, untuk merubah *keiyoushi* (misal “kawaii”) menjadi bentuk negatifnya, akhiran “i” diganti dengan “kunai”. Berarti aturan tersebut berlaku untuk kelas kata EDICT adj. Maka penanda_sumber adalah:

adj

penanda_tujuan menunjukkan kelas kata suatu kata setelah diinfleksi oleh aturan tersebut. Formatnya sama seperti penanda_sumber, yang berarti jumlahnya bisa satu atau lebih¹¹. Untuk kedua infleksi negatif seperti yang dicontohkan sebelumnya, penanda_tujuan adalah:

neg

¹¹ Contoh aturan yang memiliki penanda target lebih dari 1 adalah: v5s,v5k,v5k-s,v5g,v5b,v5t,v5m,v5r,v5aru,v5n,v5u,v5u-s/*u/*i/stem,n
Ini karena hasil infleksinya bisa berfungsi sebagai *stem* (misal untuk selanjutnya digabung dengan “masu”) maupun sebagai nomina (misal untuk diberi akhiran “da”)

Perlu dicatat bahwa *neg* bukanlah penanda yang ada pada EDICT, namun nama penanda yang dibuat sendiri. Perlu dicatat juga bahwa *penanda_sumber* secara implisit dibawa menjadi *penanda_tujuan*. Kedua penanda tersebut adalah salah satu dasar perantaraan.

pola_sumber digunakan untuk mengetahui bagian belakang kata yang diganti pada infleksi, jika ada. Formatnya untuk kasus sederhana adalah:

```
*[akhiran]
```

Sebagai contoh untuk merubah *keiyoushi* (misal “kawaii”) menjadi bentuk negatifnya, akhiran “i” diganti dengan “kunai”. Maka *pola_sumber* pada kasus tersebut adalah:

```
*i
```

Sebagai contoh lain, untuk merubah *nomina* (misal “aidoru”) dan *keiyoudoshi* (misal “suki”) menjadi bentuk negatifnya, ditambahkan “janai” di belakang kata tersebut. Berarti tidak ada yang diganti pada kata sumbernya. Maka *pola_sumber* adalah:

```
*
```

Jika terdapat perubahan suara pada *kanji* dasarnya, maka formatnya adalah:

```
*([suara_kanji])[akhiran_kana]
```

Sebagai contoh, pada perubahan 来る “kuru” menjadi 来た “kita”, suara “ku” bukanlah merupakan *okurigana* tapi merupakan suara dari *kanji*-nya. Yang merupakan *okurigana* hanyalah “ru”. Dalam kasus ini, *pola_sumber* adalah:

```
*(ku)ru
```

pola_tujuan menunjukkan tambahan di awal dan pergantian di belakang yang terjadi pada infleksi. Formatnya untuk kasus sederhana adalah:

```
[awalan]*[akhiran]
```

Sebagai contoh untuk merubah *keiyoushi* (misal “kawaii”) menjadi bentuk negatifnya, akhiran “i” diganti dengan “kunai”. Maka pola_tujuan pada kasus tersebut adalah:

```
*kunai
```

Jika terdapat perubahan suara pada *kanji* dasarnya, maka formatnya adalah:

```
*([suara_kanji])[akhiran_baru]
```

Sebagai contoh, pada perubahan 来る “kuru” menjadi 来た “kita”, suara “ki” bukanlah merupakan *okurigana* tapi merupakan suara dari *kanji*-nya. Yang merupakan *okurigana* hanyalah “ta”. Dalam kasus ini, pola_tujuan adalah:

```
*(ki)ta
```

Jika terdapat *kanji* pada infleksinya, maka *kanji* tersebut ditandai dengan format berikut¹²:

```
([romaji]||[kanji])
```

Contohnya, pada infleksi bentuk -te seperti 持って “mot'te” menjadi 持っていく”mot'teiku” atau dengan kanji 持つて行く, pola_tujuan adalah:

```
*(i|行)ku
```

Sebagai contoh lain, pada penambahan “dai” (besar) pada nomina (misal “bouken”), pola_tujuan adalah:

¹² Infleksi *kanji* di akhir hanya bisa muncul langsung setelah tanda bintang, jika tidak maka kanjinya diabaikan.

```
(dai|大) *
```

Inilah contoh suatu aturan lengkap:

```
n,adj-na/*/*janai/neg
```

Aturan tersebut mengubah nomina (*n*) dan *keiyoudoushi* (*adj-na*) menjadi bentuk negatifnya (*neg*) dengan cara menambahkan “janai” pada akhir kata.

Seluruh aturannya disimpan dalam berkas `rules_romaji.txt`. Dengan begitu, penambahan aturan menjadi sangat mudah karena hanya tinggal mengubah berkas tersebut.

3.3.4.5 Aplikasi Aturan pada Kata Nyata

Dalam proses transliterasi terdapat objek `RuleInstance` yang berbeda dengan objek `Rule`. Kalau `Rule` menyatakan aturan tata bahasa, `RuleInstance` adalah aplikasi nol atau lebih aturan tata bahasa terhadap suatu kata tertentu.

Contoh yang paling sederhanya adalah saat belum ada aturan apapun yang diaplikasikan. Misal pengguna memasukkan “aidoru”. Dalam algoritma *depth-first search* transliterasi (lihat bagian 3.3.4.7), akan dibuat objek `RuleInstance` untuk menjadi *node* pertama yang mengatakan:

```
mengasumsikan ada kata “aidoru”, kata tersebut dapat diubah menjadi “aidoru”
```

Dalam kasus tersebut belum ada `Rule` yang diterapkan, sehingga `RuleInstance` tersebut tidak merubah katanya sama sekali. `RuleInstance` tersebut dinamakan `RuleInstance` elementer.

Contoh berikutnya adalah `RuleInstance` yang dihasilkan dari penerapan satu aturan. Misal pengguna memasukkan “kawaikunai”. `RuleInstance` elementernya akan mengatakan:

mengasumsikan ada kata “kawaikunai”, kata tersebut dapat diubah menjadi “kawaikunai”

Berikutnya, mesin pentransliterasi akan berusaha mencari suatu `Rule` yang cocok, dalam artian pola tujuan `Rule` tersebut cocok dengan “kawaikunai”. Ini disebut perantaraan dan akan dijelaskan dengan penuh di bagian berikutnya (3.3.4.6). Ternyata `Rule` berikut cocok:

`adj/*i/*kunai/neg`

Dengan suatu algoritma perantaraan, maka akan dihasilkan `RuleInstance` berikut:

mengasumsikan ada kata bertipe `adj` “kawaii”, kata tersebut dapat diubah menjadi “kawaikunai”

Bisa dilihat kontrasnya, bahwa `Rule` mendeskripsikan aturan sedangkan `RuleInstance` adalah aplikasi aturan tersebut terhadap kata tertentu (dalam hal ini “kawaii”).

Pada contoh sebelumnya, “kawaii” dikatakan sebagai kata dasar dan “kawaikunai” dikatakan sebagai kata infleksi.

Inilah contoh lain suatu `RuleInstance`:

mengasumsikan ada kata bertipe `te` “mot'te”, kata tersebut dapat diubah menjadi “mot'teikimasu”

Contoh tersebut dinamakan `RuleInstance` perantara, sebab `te` bukanlah merupakan kelas kata EDICT sehingga tidak mungkin ada di basis data.

Contoh tersebut juga merupakan `RuleInstance` yang diperoleh dari penerapan 3 Rule, yaitu berturut-turut:

- `te/*/*(i|行)ku/v5k-s`
- `v5k-s/*u/*i/stem,n`
- `stem/*/*masu/masu`

Deskripsi lengkap mengenai `RuleInstance` bisa dilihat di 3.3.10.

3.3.4.6 Algoritma Perantaraan

Perantaraan adalah dibentuknya suatu `RuleInstance` baru dari `RuleInstance` yang sudah ada dan suatu Rule tertentu. `RuleInstance` baru tersebut akan memiliki kata dasar yang berbeda dengan `RuleInstance` yang sudah ada, namun kata infleksinya sama.

Sebagai contoh, misal diberikan serangkaian Rule berikut:

- a) `v5s/*u/*i/stem`
- b) `stem/*/*masu/masu`
- c) `masu/*su/*sita/past`

Dengan tiga Rule tersebut, suatu bisa diketahui bahwa suatu kata dasar `v5s` “hanasu” dapat berturut-turut diubah menjadi “hanasi” (stem), “hanasimasu” (masu), dan terakhir “hanasimasita” (past).

Perantaraan adalah proses kebalikannya. Misal pengguna memasukkan kata “hanasimasita”. Dapat dibuat `RuleInstance` elementer yang infleksinya adalah

“hanasimasita” (any) -> “hanasimasita”. Dengan merantainya menggunakan aturan c, akan diperoleh `RuleInstance` yang infleksinya “hanasimasu” (masu) -> “hanasimasita”. Dengan perantainya selanjutnya menggunakan aturan b, akan diperoleh `RuleInstance` “hanasi” (stem) -> “hanasimasita”. Perantainya terakhir dengan aturan a memberikan `RuleInstance` yang mengatakan:

mengasumsikan ada kata bertipe v5s “hanasu”, kata tersebut dapat diubah menjadi “hanasimasu”

Algoritma perantainya terdapat dalam metode `GetNextInstances` pada kelas `KanjiTransliterator`. Metode ini menerima suatu `RuleInstance` *instance* dan akan mengembalikan seluruh `RuleInstance` yang mungkin dirantainya. Di bagian paling luar metode ini adalah *loop* yang mencoba merantainya seluruh `Rule` yang telah dimuat. Akan dijelaskan algoritma di dalam *loop* tersebut, di mana `Rule` yang sedang diinvestigasi adalah variabel yang bernama *rule* dan `RuleInstance` hasil perantainya, bila ada, bernama *nextInstance*.

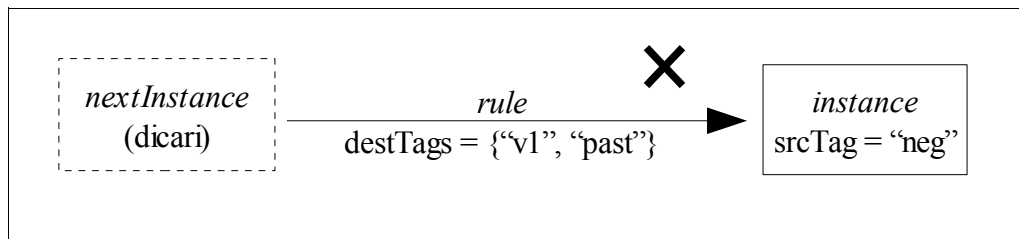
```
if(instance.srcTag != "any" &&
    Array.IndexOf(rule.destTags, instance.srcTag) == -1)
{
    continue;
}
```

Gambar 3.18 Algoritma Perantainya bagian 1

Pertama dicek apakah salah satu penanda target pada *rule* sama dengan penanda pada *instance*. Jika tidak, maka aturannya tidak boleh disambung dan *rule* berikutnya diproses (menggunakan *statement continue*). Perkecualiannya adalah jika penanda pada *instance* adalah “any” yang berarti dia bisa berlaku

sebagai kelas kata apapun.

Ilustrasi dari sebuah kasusnya adalah sebagai berikut:



Gambar 3.19 Ilustrasi Penanda yang tidak Cocok dalam Perantaraan

Pada kasus tersebut, *rule* mengatakan bahwa hasil infleksi adalah “v1” atau “past”. Namun karena penanda pada *instance* adalah “neg”, maka aturan tidak mungkin dirantai.

```
if(rule.srcPatternEnd.Length == 0 &&
    (rule.destPatternStartPlain.Length +
     rule.destPatternEndPlain.Length + 1
     > instance.srcWordRomaji.Length))
{
    continue;
}
```

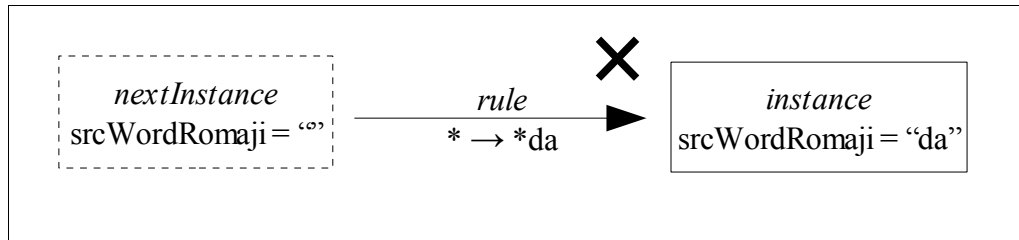
Gambar 3.20 Algoritma Perantaraan bagian 2

Berikutnya adalah pengecekan untuk aturan yang sebetulnya tidak menginfleksi namun hanya menambahkan, seperti:

```
n/*/*da/decl
```

Pengecekan yang dilakukan adalah agar aturan tersebut tidak bisa menjadi rantai untuk *instance* yang kata dasarnya hanyalah *string* yang ditambahkan tersebut. Untuk contoh aturan sebelumnya, pengecekan tersebut mencegah aturan tersebut dijadikan rantai untuk *instance* yang kata dasarnya “da” sebab itu berarti bahwa *RuleInstance* berikutnya akan memiliki kata dasar “” atau string kosong, yang berarti tidak ada kata dasar sama sekali. Inilah ilustrasi kasus yang

tidak diperbolehkan tersebut:

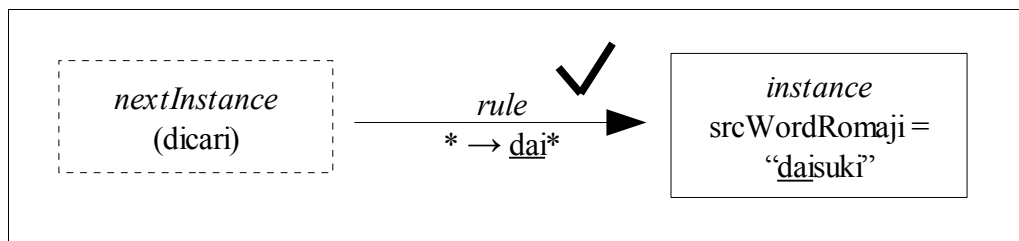


Gambar 3.21 Ilustrasi Kasus Perantaraan Kata Kosong yang Ingin Dicegah

```
if(!instance.srcWordRomaji.
    StartsWith(rule.destPatternStartPlain))
{
    continue;
}
```

Gambar 3.22 Algoritma Perantaraan bagian 3

Berikutnya adalah pengecekan apakah bagian awal dari *instance* cocok dengan pola tujuan. Inilah contoh yang cocok:

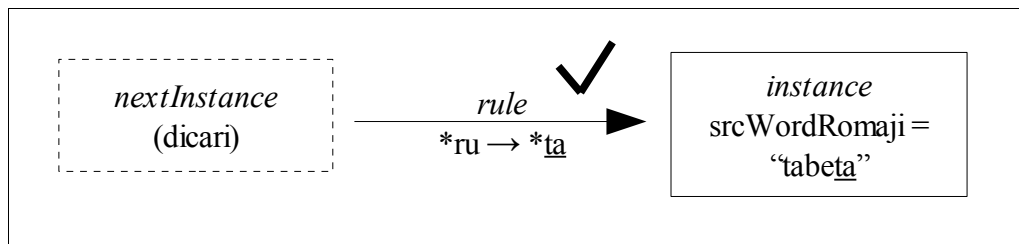


Gambar 3.23 Ilustrasi Kasus Perantaraan yang depannya Cocok

```
if(!instance.srcWordRomaji.
    EndsWith(rule.destPatternEndPlain))
{
    continue;
}
```

Gambar 3.24 Algoritma Perantaraan bagian 4

Berikutnya adalah pengecekan apakah bagian akhir dari *instance* cocok dengan pola tujuan. Inilah contoh yang cocok:



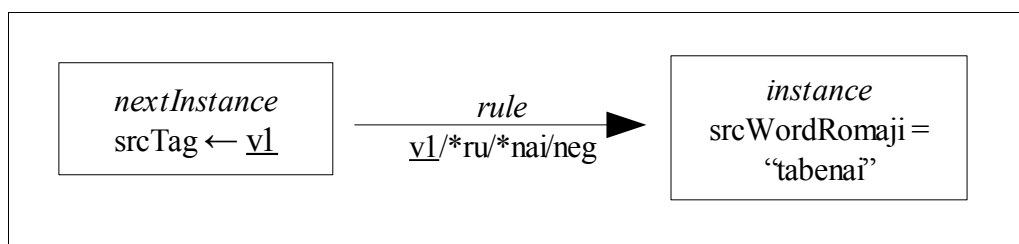
Gambar 3.25 Ilustrasi Kasus Perantain yang Belakangnya Cocok

Jika lulus semua tes di atas, artinya aturan bisa dirantai dan *nextInstance* pasti bisa ditemukan.

```
RuleInstance nextInstance = new RuleInstance();
nextInstance.srcTag = rule.srcTag;
```

Gambar 3.26 Algoritma Perantain bagian 5

Berikutnya *nextInstance* dibuat. Namun konstruktornya tidak melakukan inisialisasi apapun. Jadi selanjutnya metode ini akan mengkalkulasi nilai-nilai yang harus diisi dan mengisinya. Pada potongan kode terakhir salah satu nilai sudah diisi yaitu *srcTag*. Inilah ilustrasinya:



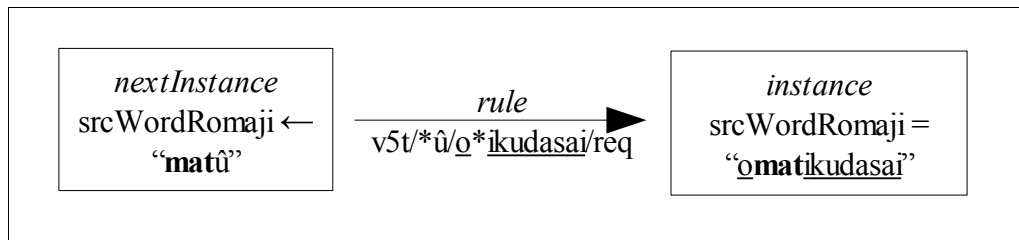
Gambar 3.27 Pemberian Nilai srcTag pada nextInstance

```
nextInstance.srcWordRomaji =
    instance.srcWordRomaji.Substring(
        rule.destPatternStartPlain.Length,
        instance.srcWordRomaji.Length -
        rule.destPatternEndPlain.Length -
        rule.destPatternStartPlain.Length) +
    rule.srcPatternEndPlain;
```

Gambar 3.28 Algoritma Perantain bagian 6

Berikutnya adalah menentukan bentuk *romaji* kata dasar dari *nextInstance*

dengan serangkaian operasi *string*. Ilustrasinya adalah sebagai berikut¹³:



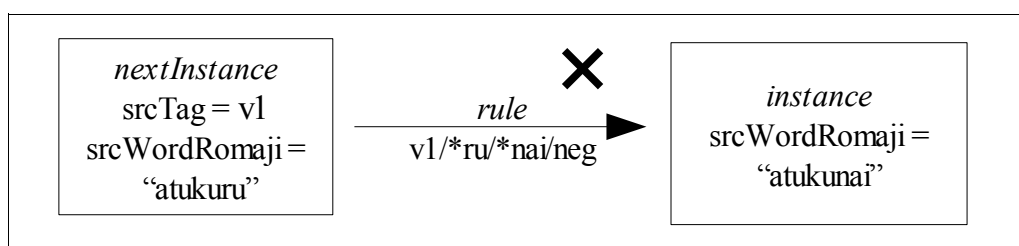
Gambar 3.29 Penentuan Kata Dasar nextInstance

```

if(!this.restrictionChecker.Check(nextInstance.srcTag,
    nextInstance.srcWordRomaji))
{
    continue;
}
  
```

Gambar 3.30 Algoritma Perantain bagian 7

Dari kata dasar dan penanda yang telah ditemukan, bisa ditentukan apakah kata dasar tersebut melanggar syarat fonologis yang ditentukan oleh penanda. Ini karena beberapa kelas kata tertentu di bahasa Jepang memiliki akhiran yang pasti. Pengecekan dilakukan oleh objek `RestrictionChecker` dan prosesnya dijelaskan secara mendetil di 3.3.12.



Gambar 3.31 Contoh Kasus yang Melanggar Syarat Fonologis Penanda

```

nextInstance.srcWordKana =
    kanaTransliteratoe.Transliterate(
        nextInstance.srcWordRomaji);
  
```

Gambar 3.32 Algoritma Perantain bagian 8

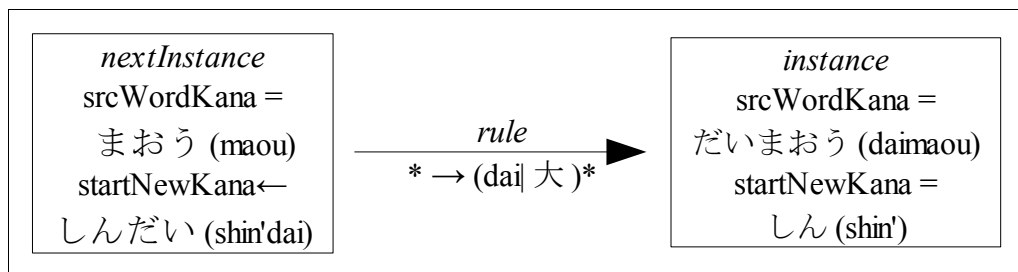
¹³ Contoh aturannya adalah rekayasa. Pada kenyataannya, di dalam kumpulan aturan yang digunakan, diperlukan 3 aturan untuk merubah “matu” ke “omatikudasai” yaitu “v5t/*u/*i/stem”, “stem/*/(o|御)*/hon”, dan “stem/*/*kudasai/req”

Setelah dipastikan tidak terjadi pelanggaran fonologis, berikutnya kata dasar *romaji*-nya ditransliterasi ke *kana*. Bentuk *kana*-nya akan digunakan untuk keperluan perantaraan dan nantinya digunakan untuk *query* ke basis data. Yang melakukan transliterasi romaji ke kana adalah objek *RomajiToKanaTransliterator* dan prosesnya dijelaskan secara mendetil di 3.3.5.

```
string insertedStartNewKana =
    kanaTransliterator.Transliterate(
        rule.destPatternStartPlain);
nextInstance.startNewKana =
    instance.startNewKana + insertedStartNewKana;
```

Gambar 3.33 Algoritma Perantaraan bagian 9

Berikutnya adalah perantaraan bagian depan *kana*-nya, yang merupakan bagian depan yang ditentukan oleh *rule* dan bagian depan selanjutnya yang ditentukan oleh *instance*.



Gambar 3.34 Contoh Perantaraan Bagian Depan Kana

```
if(instance.startNewKanji != null)
{
    nextInstance.startNewKanji = instance.startNewKanji +
        kanaTransliterator.Transliterate(
            GetKanjiPattern(rule.destPatternStart));
}else if(rule.destPatternStart.Length !=
    rule.destPatternStartPlain.Length)
{
    nextInstance.startNewKanji = instance.startNewKana +
        kanaTransliterator.Transliterate(
            GetKanjiPattern(rule.destPatternStart));
}
```

Gambar 3.35 Algoritma Perantaraan bagian 10

Berikutnya adalah perantaraan bagian depan *kanji*-nya, jika memang diperlukan. Perantaraan *kanji* depan akan dilakukan jika:

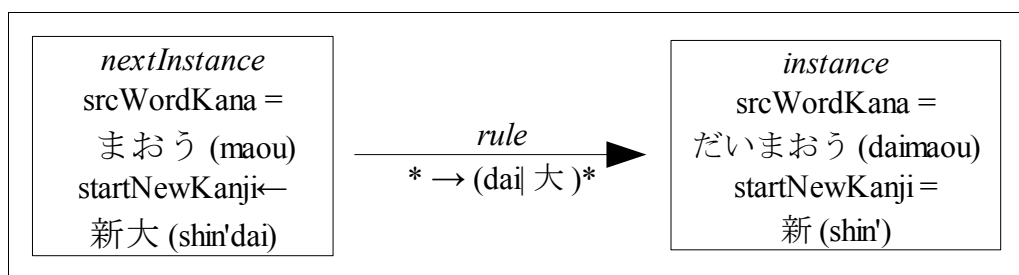
- `instance.startNewKanji != null`

Ini berarti pada *instance* sudah terdapat infleksi depan *kanji*, jadi mau tidak mau pada perantaraan informasi tersebut harus dibawa.

- `rule.destPatternStart.Length != rule.destPatternStartPlain.Length`

Ini berarti pada infleksi depan *rule* ini terdapat infleksi *kanji*. Ini dilakukan dengan membandingkan panjang `destPatternStart` dengan `destPatternStartPlain`, sebab jika terdapat spesifikasi infleksi *kanji* maka pasti `destPatternStartPlain` lebih pendek. Contohnya adalah “(dai|大)” yang pada bentuk *plain* menjadi “dai”.

Bentuk *kanji*-nya diperoleh dengan metode statik `KanjiTransliteratort.GetKanjiPattern`.



Gambar 3.36 Contoh Perantaraan Bagian Depan *Kanji*

```

int oldInstDiffIndex = instance.srcWordKana.Length -
instance.endOrigKana.Length;

int differStart = 0;
int instDiffIndex = insertedStartNewKana.Length;

```

```

for(;differStart < nextInstance.srcWordKana.Length &&
    instDiffIndex < oldInstDiffIndex &&
    nextInstance.srcWordKana[differStart] ==
    instance.srcWordKana[instDiffIndex];
    differStart++, instDiffIndex++)
{
}

```

Gambar 3.37 Algoritma Perantaraan bagian 11

Berikutnya adalah perantaraan bagian belakang. Untuk bagian belakang, prosesnya tidak sesederhana bagian depan sebab bagian belakang bisa melibatkan infleksi, tidak hanya penambahan.

Penggunaan *romaji* yang tertulis pada *rule* tidak bisa dilakukan sebab ada kemungkinan bahwa vokal yang tertulis pada awal aturan sebetulnya merupakan bagian dari *kana* lain. Contohnya adalah aturan berikut:

```
v5s,v5k,v5k-s,v5g,v5b,v5t,v5m,v5r,v5aru,v5n,v5u,v5u-s/*u/*eru/v1,pot
```

Pada aturan tersebut sebetulnya yang diinfleksi belum tentu “u” namun bisa saja misalnya “su”. Jadi yang dilakukan potongan kode terakhir adalah menentukan di mana sebetulnya infleksi belakang terjadi dengan membandingkan `nextInstance.srcWordKana` dengan `instance.srcWordKana`.

Pada kode terakhir `differStart` akan beriterasi pada `nextInstance.srcWordKana` dan `instDiffIndex` pada `instance.srcWordKana`. `oldInstDiffIndex` digunakan agar infleksi pada *instance* ikut diperhitungkan¹⁴.

```

bool soundMorph = rule.srcPatternEnd.StartsWith("(");
nextInstance.soundMorph = soundMorph;

```

Gambar 3.38 Algoritma Perantaraan bagian 12

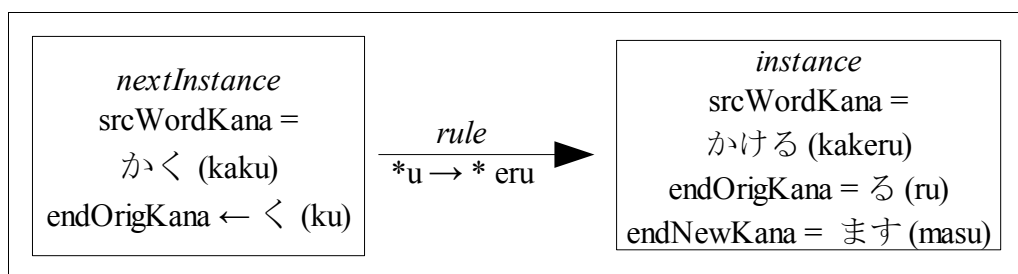
¹⁴ Sebagai contoh, jika *instance* adalah “ださなければ” yang diinfleksi menjadi “ださなきゃ”, maka pada *instance* infleksi dimulai dari け. Jika kata dasar *nextInstance* adalah “ださなける” maka perubahan juga harus dianggap berasal dari け, bukan dari る.

Setelah indeks perbedaannya ditemukan, berikutnya dicek apakah terjadi perubahan suara pada *kanji* kata dasar. Cukup dicek keberadaan kurung buka sebab itu adalah cara satu-satunya untuk menspesifikasikan perubahan suara *kanji* dasar.

```
nextInstance.endOrigKana =
    nextInstance.srcWordKana.Substring(differStart);
```

Gambar 3.39 Algoritma Perantain bagian 13

Setelah mengetahui indeks perubahannya, berikutnya *endOrigKana* dari *nextInstance* bisa dengan mudah dicari. Inilah ilustrasinya:

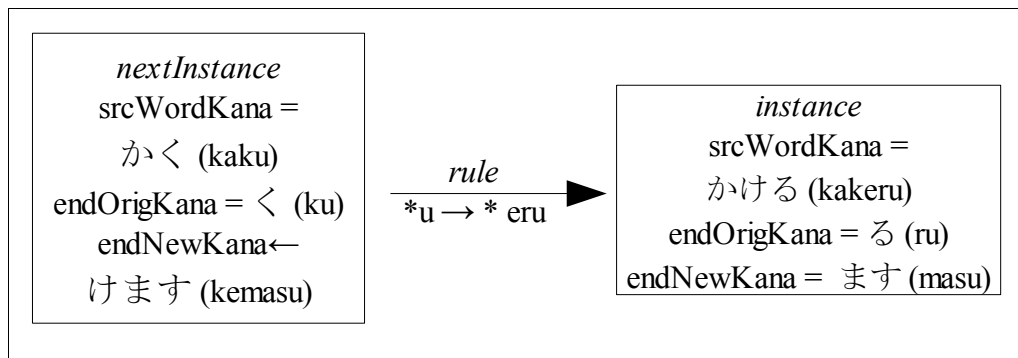


Gambar 3.40 Contoh Penentuan Nilai *endOrigKana* pada Perantain

```
nextInstance.endNewKana =
    instance.srcWordKana.Substring(instDiffIndex);
nextInstance.endNewKana =
    nextInstance.endNewKana.Substring(
        0, nextInstance.endNewKana.Length -
        instance.endOrigKana.Length);
nextInstance.endNewKana += instance.endNewKana;
```

Gambar 3.41 Algoritma Perantain bagian 14

Berikutnya adalah penentuan *endNewKana*. Pertama ditentukan bentuk infleksinya berdasarkan *rule*, lalu dirantain dengan infleksi pada *instance*. Inilah ilustrasinya:



Gambar 3.42 Contoh Penentuan Nilai endNewKana pada Perantaraan

```

bool kanjiReplacement =
    !soundMorph && rule.destPatternEnd.StartsWith("(");

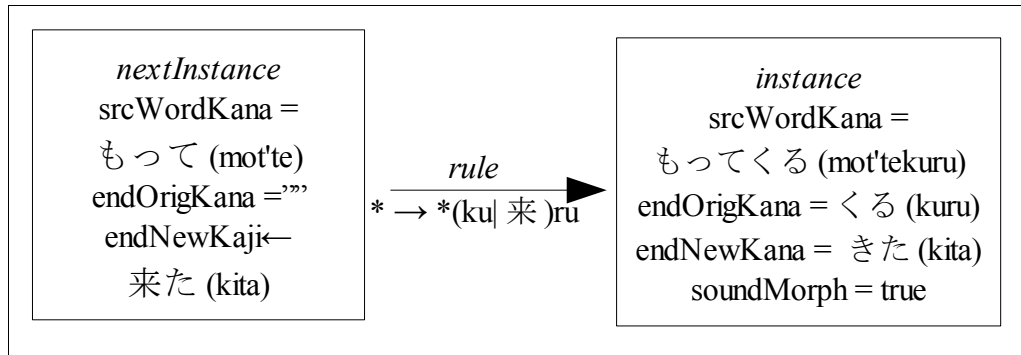
if(kanjiReplacement)
{
nextInstance.endNewKanji =
    kanaTransliterater.Transliterate(
        GetKanjiPattern(rule.destPatternEnd));

if(instance.soundMorph)
{
    soundMorphModifier = 1;
}
nextInstance.endNewKanji =
    nextInstance.endNewKanji.Substring(
        0, nextInstance.endNewKanji.Length -
        instance.endOrigKana.Length +
        soundMorphModifier);
string addition = instance.endNewKanji == null ?
    instance.endNewKana : instance.endNewKanji;
nextInstance.endNewKanji +=
    addition.Substring(soundMorphModifier);
}
  
```

Gambar 3.43 Algoritma Perantaraan bagian 15

Berikutnya adalah penentuan *endNewKanji*. Kasus pertamanya adalah kalau memang pada *nextInstance* didefinisikan infleksi *kanji* (pengecekan if yang pertama). Harus dicek apakah infleksi pada *instance* mengandung *soundMorph*, agar *kanji* yang bersangkutan tidak hilang karena diinfleksi menjadi *kana*.

Ilustrasi contohnya adalah sebagai berikut:



Gambar 3.44 Contoh Penentuan Nilai endNewKanji pada Perantaraan karena rule

```

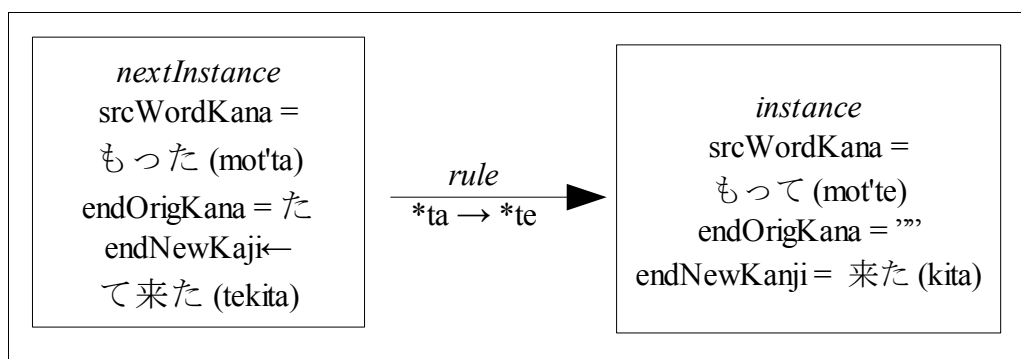
else if(instance.endNewKanji != null)
{
    nextInstance.endNewKanji =
        instance.srcWordKana.Substring(instDiffIndex);

    nextInstance.endNewKanji =
        nextInstance.endNewKanji.Substring(0,
            nextInstance.endNewKanji.Length -
            instance.endOrigKana.Length);
    nextInstance.endNewKanji += instance.endNewKanji;
}

```

Gambar 3.45 Algoritma Perantaraan bagian 16

Terakhir adalah penentuan *endNewKanji* karena terdapat infleksi kanji pada *instance*, bukan pada *rule*. Ilustrasi contohnya adalah sebagai berikut:



Gambar 3.46 Contoh Penentuan Nilai endNewKanji pada Perantaraan karena instance

Dengan langkah terakhir tadi, dua buah `RuleInstance` berhasil diratai.

3.3.4.7 Algoritma Breadth-first Search

Algoritma ini menerima masukan berupa *string inputRomaji* dan mengembalikan sejumlah `RuleInstance` non-perantara yang berisi semua kemungkinan kata yang memiliki rantai infleksi yang hasil akhirnya adalah *inputRomaji*.

Algoritma ini bekerja dengan prinsip breadth-first search, dengan *node* awalnya berupa `RuleInstance` elementer. Dari *node* awal tersebut dilakukan pencarian secara mundur sehingga pada akhirnya kata dasar yang bersesuaian akan ditemukan.

Inilah pseudocode dari algoritma tersebut:

1. Inisialisasi *List* keluaran dan antrian *breadth-first search*
2. Buat `RuleInstance` elementer dari *inputRomaji* dan masukkan `RuleInstance` tersebut ke *List* dan antrian.
3. Selama antrian belum kosong
 - a) Ambil `RuleInstance` *ibu* di awal antrian
 - b) Untuk tiap `RuleInstance` *anak* yang diperoleh menggunakan metode perantaraan `GetNextInstances` (3.3.4.6)
 - i. Masukkan *anak* ke antrian
 - ii. Jika *anak* bukan `RuleInstance` perantara, masukkan ke *List* keluaran
4. Kembalikan *List* keluaran

Dari *List* keluaran yang diperoleh, selanjutnya bisa dilakukan pencarian ke basis data terhadap kata dasarnya.

Implementasi yang sebenarnya melakukan sedikit modifikasi terhadap bagaimana keluarannya diorganisir. Himpunan `RuleInstance` yang dikembalikan sama, hanya saja tidak diorganisir dalam *List* sederhana namun dikelompokkan lagi berdasarkan ketinggiannya pada pohon pencarian¹⁵ (3.3.13.5).

3.3.5 Mesin Transliterasi Romaji ke Kana dan Sebaliknya

3.3.5.1 Tinjauan

Dalam proses transliterasi *kanji* yang dilakukan `GamaIme.dll`, terdapat tahap yang memerlukan transliterasi dari *romaji* ke *kana* dan sebaliknya. Transliterasi ini relatif sederhana dibandingkan dengan transliterasi *kanji* karena:

- Jumlah karakter *kana* terbatas, tidak seperti *kanji* yang jumlah karakternya puluhan ribu.
- Transliterasinya merupakan pemetaan *string* ke *string* yang sederhana. Sebagai contoh, “i” jika ditransliterasi menjadi *hiragana* pasti menjadi `い`. Ini beda dengan transliterasi *kanji* di mana satu bunyi bisa berkorespondensi dengan banyak *kanji*. Sebagai contoh, bunyi “i” berkorespondensi dengan lebih dari 100 *kanji* (囲, 位, 以, 胃, dan lain-lain).

3.3.5.2 Algoritma

¹⁵ Hasil pada tingkat pohon yang lebih atas dianggap lebih relevan karena membutuhkan lebih sedikit infleksi, contohnya adalah *karada* (体, tidak ada infleksi) dengan *kara da* (空だ, 1 infleksi)

Pertama akan diberikan algoritma untuk transliterasi *romaji* ke *kana*. Secara matematis, didefinisikan fungsi transliterasi $T : S \rightarrow D$ dengan S himpunan *string* sumber (source) dan D himpunan *string* hasil (destination). Sebagai contoh, $T("i") = "い"$, $T("ma") = "ま"$, dan $T("kya") = "きや"$.

Dalam kasus transliterasi *romaji* ke *kana*, fungsinya tidak perlu invertibel untuk mengakomodasi lebih dari satu sistem romanisasi. Sebagai contoh, $T("shi")$ $T("si") = "し"$ sehingga romanisasi Hepburn maupun Nihon-shiki bisa digunakan.

Untuk mendapatkan katakana, digunakan huruf besar. Sebagai contoh, $T("a") = "あ"$ (hiragana) sedangkan $T("A") = "ア"$ (katakana).

Pemetaan fungsi T sepenuhnya ada di `romaji_to_hiragana.txt` dan `romaji_to_katakana.txt` pada lampiran.

Dengan fungsi T tersebut dan *string* masukan *input*, algoritmanya adalah:

1. Inisialisasi *string* keluaran *output* sebagai *string* kosong
2. Untuk $i = 0$ sampai $\text{panjang}(\text{input}) - 1$
 1. Cari s anggota S yang merupakan *substring* terpanjang pada *input* dimulai dari indeks i
 2. Jika ada, tambahkan $T(s)$ pada *output*, dan naikkan i sejumlah $\text{panjang}(s)$. Ini adalah kasus di mana terdapat transliterasinya pada posisi i .
 3. Jika tidak, tambahkan $\text{input}[i]$ pada *output*, dan naikkan i sejumlah 1. Ini berarti pada posisi i tidak ada transliterasinya sehingga karakter di *string* masukkan disalin saja ke keluaran.
3. Hasil transliterasi adalah *output*

Secara singkat, algoritmanya memindai *string* masukan secara linier dari kiri ke kanan dan mencoba melakukan pencocokan *greedy* atau terpanjang untuk ditransliterasi.

Diberikan contoh transliterasi untuk *string* masukan “kinen” (perayaan):

1. $output = ""$, indeks $i = 0$ yang menunjuk pada karakter “k” (“kinen”)
2. Pada posisi tersebut, anggota S satu-satunya yang merupakan *substring* input adalah “ki”. Maka pada *output* ditambahkan $T(“ki”) = “き”$. *output* menjadi “き”.
3. Indeks i ditambahkan sebanyak panjang dari “ki” yaitu 2. Sekarang $i = 2$ yang menunjuk karakter “n” pertama (“kinen”).
4. Pada posisi tersebut, anggota S yang merupakan *substring* adalah “n” dan “ne”. Karena yang terpanjang adalah “ne”, maka itulah yang akan ditransliterasi. Pada *output* ditambahkan $T(“ne”) = “ね”$. *output* menjadi “きね”.
5. Indeks i ditambahkan sebanyak panjang dari “ne” yaitu 2. Sekarang $i = 4$ yang menunjuk karakter “n” kedua (“kinenn”).
6. Pada posisi tersebut, anggota S satu-satunya yang merupakan *substring* input adalah “n”. Maka pada *output* ditambahkan $T(“n”) = “ん”$. *output* menjadi “きねん”.
7. Indeks i ditambahkan sebanyak panjang dari “n” yaitu 1. Sekarang $i = 5$.
8. Karena indeks sudah menunjuk ke luar *string input*, maka algoritma selesai. *output* adalah “きねん”.

Untuk transliterasi dari *kana* ke *romaji*, yang beda hanyalah fungsi

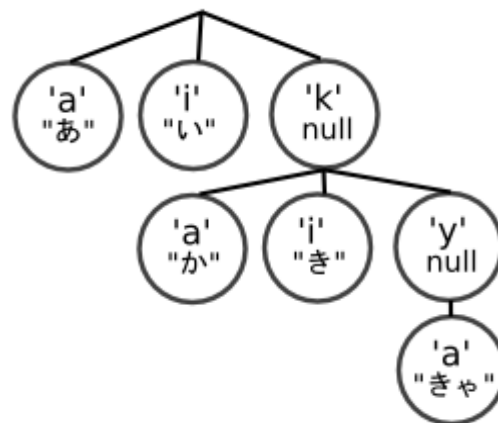
transliterasinya. Untuk mengetahui pemetaannya, bisa dilihat hiragana_to_romaji.txt pada lampiran, yang sama persis dengan deskripsi romanisasi GamaIme.dll pada 3.3.4.3.

3.3.5.3 Implementasi

Secara internal, aturan direpresentasikan dalam bentuk pohon. Sebagai contoh, dengan aturan berikut:

```
"a" -> "あ"
"i" -> "い"
"ka" -> "か"
"ki" -> "き"
"ky" -> "きゃ"
```

Maka pohon yang dihasilkan adalah:

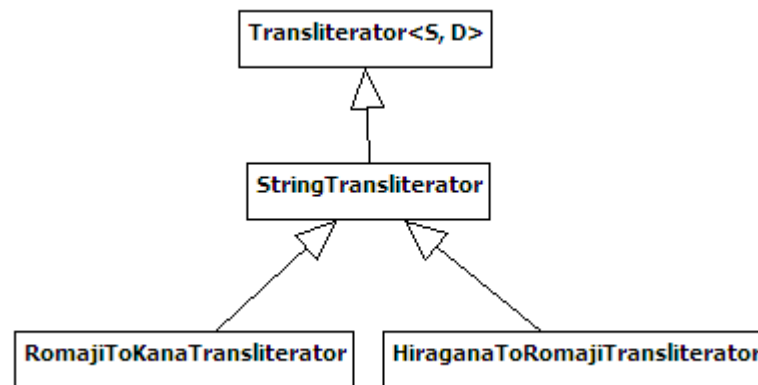


Gambar 3.47 Contoh Pohon Aturan Transliterasi Kana

Dari situ bisa terlihat bahwa pola masukan “k” dan “ky” tidak akan diterima.

Kelas-kelas yang terlibat dalam transliterasi *kana* ini adalah Transliterators, dan StringTransliterators,

RomajiToKanaTransliteratur, dan HiraganaToRomajiTransliteratur. Hubungannya adalah sebagai berikut:



Gambar 3.48 Diagram Hubungan berbagai Kelas Pentransliterasi Kana
Pembahasan lebih lanjut ada di subsubbab masing-masing kelas.

3.3.6 Kelas `Transliteratur<S, D>`

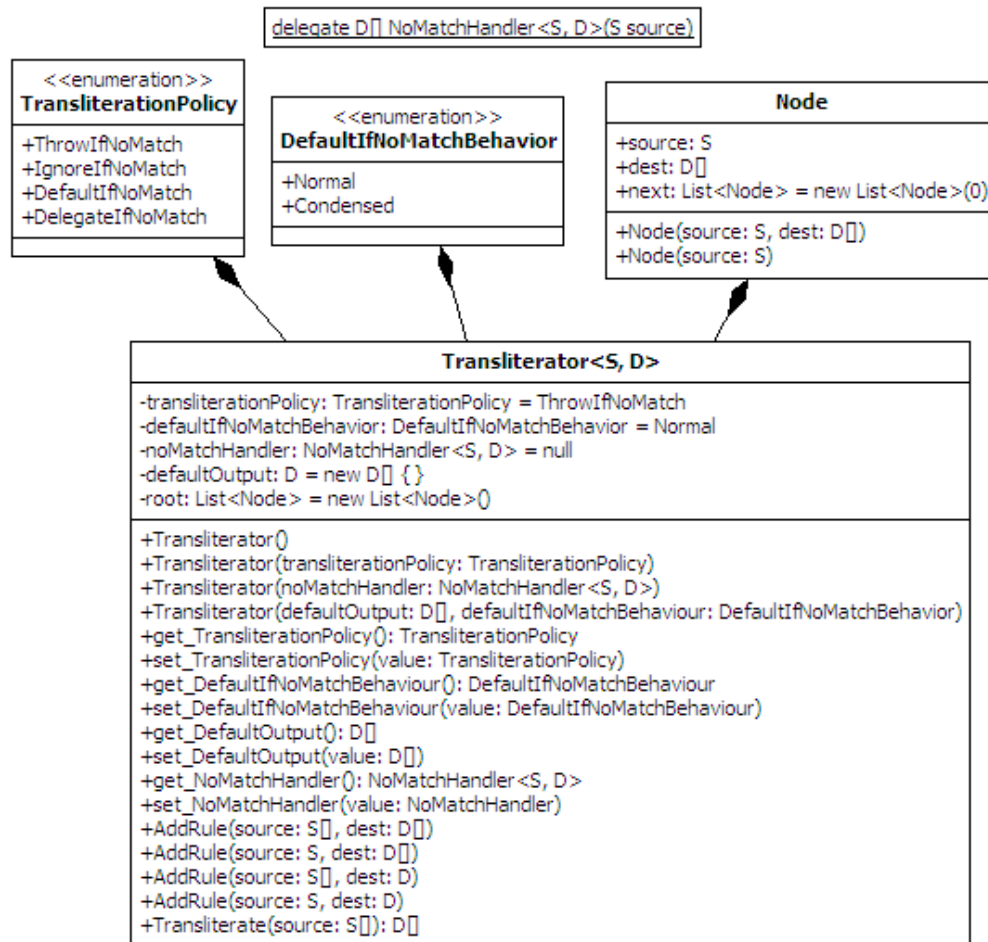
3.3.6.1 Tinjauan

Kelas `Transliteratur` berfungsi mengubah suatu larik bertipe *S* menjadi larik bertipe *D*. Karena diimplementasikan menggunakan *generics*, pengguna kelas ini bebas menentukan tipe *S* dan *D* saat membuat suatu objek `Transliteratur`. Penjelasan mengenai algoritma transliterasinya pada *string*, yang bisa digeneralisasi untuk sembarang tipe *S* dan *D*, ada pada 3.3.5.

3.3.6.2 Perancangan Kelas

Diagram kelas `Transliteratur` beserta kelas dan enumerasi yang

berhubungan adalah sebagai berikut:



Gambar 3.49 Diagram Kelas Transliteratort

3.3.6.3 Penggunaan

Contoh berikut membuat suatu objek Transliteratort yang dapat mengubah larik karakter menjadi larik karakter:

```
Transliteratort<char, char> t = new Transliteratort<char, char>();
```

Transliteratort bekerja dengan seperangkat aturan. Aturan adalah

pasangan larik S (disebut pola masukan) dengan larik D (disebut pola keluaran). Transliterator akan mengubah kemunculan S pada masukan menjadi D pada keluaran. Dalam proses perubahan, Transliterator akan mencari aturan dengan pola masukan terpanjang yang cocok.

Sebagai contoh, misal pada objek Transliterator sebelumnya ingin diberikan aturan sebagai berikut:

```
{'a'} -> {'A'}
{'b'} -> {'B'}
{'c'} -> {'C'}
{'b', 'a'} -> {'X', 'Y', 'Z'}
```

Caranya adalah dengan memanggil metode `AddRule`:

```
t.AddRule("a".ToCharArray(), "A".ToCharArray());
t.AddRule("b".ToCharArray(), "B".ToCharArray());
t.AddRule("c".ToCharArray(), "C".ToCharArray());
t.AddRule("ba".ToCharArray(), "XYZ".ToCharArray());
```

Untuk mengubah suatu larik S menjadi keluaran berdasarkan aturan yang diberikan, metode yang digunakan adalah `Transliterate`. Contohnya adalah:

```
char[] output = t.Transliterate("caca".ToCharArray());
```

Pada objek Transliterator contoh, inilah keluaran metode `Transliterate` untuk beberapa contoh masukan:

Masukan	Keluaran
{'c', 'a', 'c', 'a'}	{'C', 'A', 'C', 'A'}
{'a', 'b', 'c', 'a'}	{'A', 'B', 'C', 'A'}
{'b', 'a', 'c', 'a'}	{'X', 'Y', 'Z', 'C', 'A'}
{'a', 'b', 'a', 'b'}	{'A', 'X', 'Y', 'Z', 'B'}

Tabel 3.2 Contoh Masukan dan Keluaran Transliterasi Karakter

{'b', 'a', 'c', 'a'} diubah menjadi {'X', 'Y', 'Z',

'C', 'A'} sebab aturan {'b', 'a'} -> {'X', 'Y', 'Z'}-lah yang terpilih, bukan {'b'} -> {'B'} disusul {'a'} -> {'A'}. Ini karena pola masukan {'b', 'a'} lebih panjang dari pola masukan {'b'} (2 elemen lawan 1 elemen).

Secara default, `Transliterate` akan melempar eksepsi jika tidak ditemukan pola masukan yang cocok. Namun perilaku ini dapat diubah melalui properti `TransliterationPolicy` pada objek `Transliterator`. `TransliterationPolicy` merupakan suatu enumerasi dan nilai yang tersedia adalah:

- `ThrowIfNoMatch`: Melempar eksepsi. Ini perilaku default.
- `IgnoreIfNoMatch`: Elemen masukan yang bermasalah dilewati.
- `DefaultIfNoMatch`: Elemen masukan yang bermasalah diganti suatu larik *D* default yang ditentukan melalui properti `DefaultOutput`.
- `DelegateIfNoMatch`: Elemen masukan yang bermasalah dikirim ke suatu *delegate* untuk ditangani. *Delegate* bisa ditentukan melalui properti `NoMatchHandler`.

Delegate yang digunakan untuk menangani elemen masukan yang bermasalah bertipe `NoMatchHandler` yang deklarasinya adalah:

```
public delegate D[] NoMatchHandler<S, D>(S source);
```

Jika `TransliterationPolicy` adalah `DefaultIfNoMatch`, maka tersedia dua perilaku yang diatur melalui properti

DefaultIfNoMatchBehavior yang bisa bernilai:

- Normal: Tiap elemen masukan yang bermasalah diganti larik keluaran default.
- Condensed: Elemen yang bermasalah secara berurutan hanya diganti oleh satu keluaran default.

Sebagai contoh, misal terdapat objek Transliterator yang mengubah larik karakter ke larik *integer*. Lalu misal aturannya adalah:

```
{'a'} -> {1}
{'b'} -> {2}
{'c'} -> {3}
{'a', 'b'} -> {4}
default: {0}
```

Maka saat mengubah masukan {'b', 'a', 'z', 'z', 'a', 'b'}, hasilnya adalah:

Perilaku	Keluaran
TransliterationPolicy.ThrowIfNoMatch (<i>default</i>)	ArgumentException akan dilempar
TransliterationPolicy.IgnoreIfNoMatch	{2, 1, 4}
TransliterationPolicy.DefaultIfNoMatch dan DefaultIfNoMatchBehavior.Normal	{2, 1, 0, 0, 0, 4}
TransliterationPolicy.DefaultIfNoMatch dan DefaultIfNoMatchBehavior.Condensed	{2, 1, 0, 4}
TransliterationPolicy.DelegateIfNoMatch	tergantung <i>delegate</i> yang menangannya

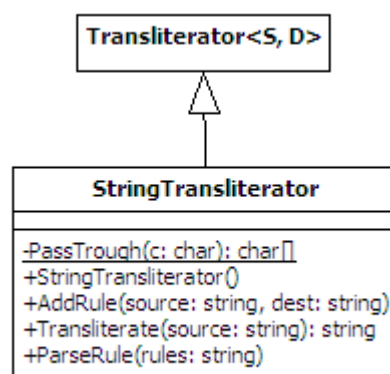
3.3.7 Kelas StringTransliterator

3.3.7.1 Tinjauan

Kelas ini merupakan spesialisasi dari kelas Transliterator, yaitu kelas Transliterator yang menangani *string*. Dengan kelas ini, pengguna tidak perlu berurusan dengan data internal dari kelas Transliterator yaitu larik karakter. Kelas ini juga menyediakan fasilitas *parsing* berkas aturan transliterasi *string*.

3.3.7.2 Perancangan Kelas

Diagram kelas dari StringTransliterator adalah sebagai berikut:



Gambar 3.50 Diagram Kelas StringTransliterator

Bisa dilihat bahwa kelas ini menyediakan metode `AddRule` dan `Transliterate` yang berurusan dengan *string*, bukan dengan larik karakter yang sebetulnya digunakan secara internal.

Konstruktor dari `StringTransliterator` mengatur agar mode *default* transliterasi menggunakan fungsi `PassTrough` jika transliterasi gagal

dilakukan pada posisi karakter tertentu. Yang dilakukan fungsi `PassTrough` tersebut hanyalah mengembalikan karakter yang gagal, sehingga pada hasil transliterasi karakter tersebut tidak berubah.

3.3.7.3 Metode Statik `ParseRule(string rules)`

Metode ini akan melakukan parsing aturan pada *string*, sehingga pengguna tidak harus secara programatis memanggil metode `AddRule` berulang-ulang untuk menambahkan sejumlah aturan.

Tiap baris pada *string* berisi aturan, dengan baris kosong diabaikan.

Format tiap barisnya adalah:

```
[sumber] [tab] [tujuan]
```

String [sumber] pada masukan akan ditransliterasi menjadi [tujuan] secara *greedy*. Inilah contoh 5 baris pertama pada `romaji_to_hiragana.txt` yang mengilustrasikan format tersebut:

```
a      あ
i      い
u      う
e      え
o      お
```

3.3.8 Kelas `RomajitoKanaTransliterator`

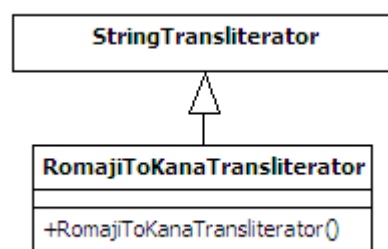
3.3.8.1 Tinjauan

Ini adalah turunan `StringTransliterator` yang konstruktornya secara otomatis memuat aturan pada `romaji_to_hiragana.txt` dan `romaji_to_katakana.txt` yang di-embed dalam `GamaIme.dll`. Dengan

memanggil metode `Transliterate` pada objek kelas ini, pengguna bisa melakukan transliterasi dari *romaji* ke *kana*.

3.3.8.2 Perancangan Kelas

Metode baru yang didefinisikan pada kelas ini hanyalah konstruktor:



Gambar 3.51 Diagram Kelas `RomajiToKanaTransliterator`

3.3.8.3 Contoh Penggunaan

Pengguna hanya perlu membuat objeknya dan memanggil metode `Transliterate`:

```

RomajiToKanaTransliterator transliterator = new
RomajiToKanaTransliterator();
transliterator.Transliterate ("watashi"); // わたし
transliterator.Transliterate ("RABU"); // ラブ
transliterator.Transliterate ("xyz"); // xyz
  
```

Gambar 3.52 Contoh Penggunaan `RomajiToKanaTransliterator`

Untuk mengetahui daftar lengkap transliterasi yang mungkin, bisa dilihat `romaji_to_hiragana.txt` dan `romaji_to_katakana.txt` di lampiran.

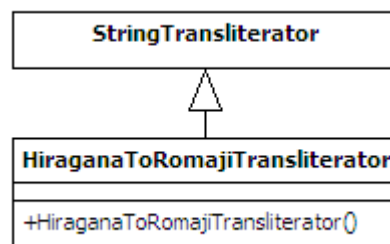
3.3.9 Kelas `HiraganaToRomajiTransliterator`

3.3.9.1 Tinjauan

Kelas turunan `StringTransliterator` ini bertujuan untuk menormalisasi penulisan *romaji* suatu *hiragana* untuk keperluan transliterasi. Konstruktornya memuat `hiragana_to_romaji.txt` yang di-embed pada `GamaIme.dll`.

3.3.9.2 Perancangan Kelas

Sebagaimana `RomajiToKanaTransliterator`, anggota baru yang didefinisikan hanyalah konstruktor.



Gambar 3.53 Diagram Kelas `HiraganaToRomajiTransliterator`

3.3.9.3 Contoh Penggunaan

Inilah contoh penggunaannya untuk menormalisasi “watashi” menjadi “watasi”:

```

RomajiToKanaTransliterator t1 = new
RomajiToKanaTransliterator();
string hiragana = t1.Transliterate("watashi"); // わたし
HiraganaToRomajiTransliterator t2 = new
HiraganaToRomajiTransliterator();
string romaji = t2.Transliterate(hiragana); // watasi
  
```

Gambar 3.54 Contoh Penggunaan `HiraganaToRomajiTransliterator`

3.3.10 Kelas RuleInstance

3.3.10.1 Tinjauan

Objek `RuleInstance` menyimpan informasi mengenai aturan infleksi yang konkrit. Secara lebih spesifik, objek ini memiliki informasi mengenai:

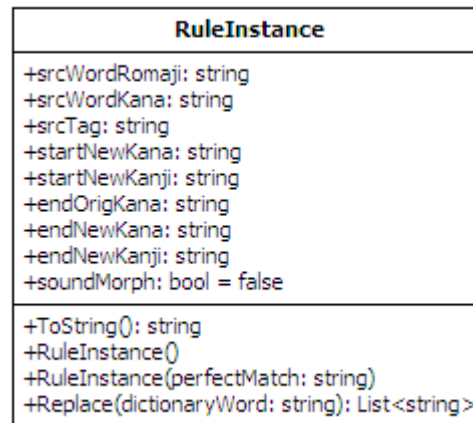
- Apa kata dasar yang harus diinfleksi.
- Apa kelas kata dari kata dasar tersebut. Objek ini mengenal kelas kata khusus “any” yang berarti bahwa tidak ada batasan mengenai kelas katanya.
- Apa yang harus ditambahkan di awal kata tersebut, kalau ada.
- Apa yang harus diubah di akhir kata tersebut, kalau ada.
- Apakah terjadi perubahan suara pada *kanji* yang bersangkutan. Ini untuk mengakomodasi beberapa infleksi tak beraturan. Contohnya adalah “kuru” (datang) menjadi “kita” (datang, bentuk lampau), yang penulisannya menggunakan kanji masing-masing adalah 来る dan 来た. Dalam contoh tersebut, suara karakter 来 berubah dari “ku” menjadi “ki”, sesuatu yang umumnya tidak terjadi pada infleksi.

Peran objek ini dalam proses transliterasi adalah:

- Menjadi suatu node dalam proses perantaraan
- Memberikan informasi mengenai kata yang harus di-*query* di basis data
- Mengubah kata yang telah diperoleh dari basis data dari bentuk dasarnya menjadi bentuk terinfleksinya

3.3.10.2 Perancangan Kelas

Diagram kelasnya adalah sebagai berikut:



Gambar 3.55 Diagram Kelas RuleInstance

Penjelasan mengenai atributnya:

- `srcWordRomaji` menyimpan kata dasar yang harus diinfleksi menggunakan *romaji*. *String* ini digunakan dalam proses perantaraan.
- `srcWordKana` menyimpan kata dasar yang harus diinfleksi menggunakan *hiragana*. *String* ini yang akan dicari di basis data.
- `srcTag` menyimpan kelas kata dari kata dasar. Ini digunakan untuk menyeleksi apakah kata yang telah didapat dari basis data bisa diinfleksi menggunakan aturan ini. Ini juga digunakan untuk menentukan apakah kata dasarnya perlu dicari di basis data, sebab terdapat bentuk-bentuk perantara yang hanya dihasilkan pada proses perantaraan namun bukan merupakan kata dasar di basis data.
- `startNewKana` menentukan apa yang harus ditambahkan di awal kata tersebut, kalau ada. Penulisannya dalam *hiragana*. Jika tidak ada maka

nilainya adalah *string* kosong. Digunakan dalam proses infleksi.

- `startNewKanji` menentukan penulisan *kanji* dari `startNewKana`, jika ada. Jika `startNewKana` kosong, atau jika tidak ada penulisan *kanji*-nya, maka nilainya *null*. Digunakan dalam proses infleksi.
- `endOrigKana` menentukan akhiran *hiragana* yang akan tergantikan pada proses infleksi. Jika tidak ada akhiran yang dihilangkan, maka nilainya adalah *string* kosong. Digunakan dalam proses infleksi.
- `endNewKana` menentukan pengganti dari `endOrigKana`, jika ada. Penulisannya dalam *hiragana*. Jika tidak ada, maka nilainya adalah *string* kosong. Jika ini tidak kosong sedangkan `endOrigKana` kosong, maka artinya adalah proses penambahan, bukan penggantian.
- `endNewKanji` menentukan bentuk *kanji* dari `endNewKana`, jika ada. Jika tidak ada, maka nilainya *null*.
- `soundMorph` menentukan apakah terjadi perubahan suara pada kanji kata dasarnya. Jika nilainya *true*, maka karakter pertama pada `endOrigKana` tidak akan digantikan oleh karakter pertama pada `endNewKana` dan `endNewKanji`.

3.3.10.3 Ilustrasi Nilai Atribut

Inilah contoh beberapa kasus infleksi dan nilai dari seluruh atributnya. Nilai-nilai tersebut diperoleh dari proses perantaraan terhadap masukan yang diberikan pengguna (dituliskan). Perlu dicatat bahwa dalam proses perantaraan, pada umumnya terdapat banyak objek `RuleInstance` lain yang dihasilkan,

namun yang ditunjukkan di sini adalah `RuleInstance` yang menghasilkan transliterasi yang diharapkan. `String` kosong ditandai dengan -.

- a) Masukan pengguna: “terebi” (televisi)

Salah satu `RuleInstance` yang dihasilkan oleh perantaraan:

Atribut	Nilai
<code>srcWordRomaji</code>	terebi
<code>srcWordKana</code>	てれび
<code>srcTag</code>	any
<code>startNewKana</code>	-
<code>startNewKanji</code>	<i>null</i>
<code>endOrigKana</code>	-
<code>endNewKana</code>	-
<code>endNewKanji</code>	<i>null</i>
<code>soundMorph</code>	<i>false</i>

Jenis kasus: tidak ada perubahan terhadap kata dasar

- b) Masukan pengguna: “tabeta” (telah makan)

Salah satu `RuleInstance` yang dihasilkan oleh perantaraan:

Atribut	Nilai
<code>srcWordRomaji</code>	taberu
<code>srcWordKana</code>	たべる
<code>srcTag</code>	v1
<code>startNewKana</code>	-
<code>startNewKanji</code>	<i>null</i>
<code>endOrigKana</code>	る
<code>endNewKana</code>	た
<code>endNewKanji</code>	<i>null</i>
<code>soundMorph</code>	<i>false</i>

Jenis kasus: infleksi di akhir

- c) Masukan pengguna: “daibouken” (petualangan besar)

Salah satu RuleInstance yang dihasilkan oleh perantaraan:

Atribut	Nilai
srcWordRomaji	bouken'
srcWordKana	ぼうけん
srcTag	n
startNewKana	だい
startNewKanji	大
endOrigKana	-
endNewKana	-
endNewKanji	<i>null</i>
soundMorph	<i>false</i>

Jenis kasus: penambahan awalan (dengan alternatif kanji)

- d) Masukan pengguna: “omatikudasai” (tolong tunggu, bentuk honorifik)

Salah satu RuleInstance yang dihasilkan oleh perantaraan:

Atribut	Nilai
srcWordRomaji	matu
srcWordKana	まつ
srcTag	v5t
startNewKana	お
startNewKanji	御
endOrigKana	つ
endNewKana	ちください
endNewKanji	<i>null</i>
soundMorph	<i>false</i>

Jenis kasus: penambahan awalan (dengan alternatif kanji) dan infleksi di akhir

- e) Masukan pengguna: “kimasita” (telah datang, bentuk sopan)

Salah satu RuleInstance yang dihasilkan oleh perantaraan:

Atribut	Nilai
srcWordRomaji	kuru
srcWordKana	くる
srcTag	vk
startNewKana	-
startNewKanji	<i>null</i>
endOrigKana	くる
endNewKana	きました
endNewKanji	<i>null</i>
soundMorph	<i>true</i>

Jenis kasus: infleksi dengan perubahan suara pada kanji

- f) Masukan pengguna: “samuitoit'ta” (dia mengatakan dingin)

Salah satu RuleInstance yang dihasilkan oleh perantaraan:

Atribut	Nilai
srcWordRomaji	samui
srcWordKana	さむい
srcTag	any
startNewKana	-
startNewKanji	<i>null</i>
endOrigKana	-
endNewKana	といた
endNewKanji	と言った
soundMorph	<i>false</i>

Jenis kasus: penambahan di akhir dengan alternatif *kanji*

3.3.10.4 Metode string RuleInstance.ToString()

Metode ini mengembalikan representasi penuh RuleInstance dalam bentuk *string*. Formatnya adalah:

```
[srcTag]:[srcWordRomaji]([srcWordKana)][[startNewKana]/
```

```
[startNewKanji]] [[endOrigKana]/[endNewKana]/[endNewKanji]]
[soundMorph]
```

Untuk string yang kosong maupun *null*, pada keluaran tersebut tidak akan dituliskan apa-apa. Jika `soundMorph` bernilai `true`, maka keluarannya adalah `M` dan jika tidak maka tidak akan dituliskan apa-apa.

Pada contoh pertama di 3.3.10.3, keluaran dari metode ini adalah:

```
any: terebi (てれび) [/] [//]
```

Metode ini digunakan saat program dikembangkan, untuk debugging.

3.3.10.5 Konstruktor `RuleInstance()`

Konstruktor ini tidak melakukan inisialisasi apapun. Ini karena pada proses transliterasi, yang mengisi nilai atribut saat proses perantaraan adalah objek `KanjiTransliterator`.

3.3.10.6 Konstruktor `RuleInstance(string perfectMatch)`

Konstruktor ini menginisialisasi suatu `RuleInstance` yang tidak melakukan infleksi apapun terhadap kata dasarnya. Contoh *instance*-nya adalah contoh pertama pada 3.3.10.3. Dalam proses perantaraan, objek `RuleInstance` tersebut menjadi *node* akar pada pohon perantaraan.

3.3.10.7 Metode `List<string> Replace(string dictionaryWord)`

Metode ini menerima masukan berupa suatu kata dasar dalam *hiragana*, *katakana*, maupun *kanji*. Dalam proses transliterasi, kata dasar tersebut diperoleh

dari *query* basis data. Yang dikembalikan metode ini adalah hasil infleksinya menurut aturan pada *instance* tersebut.

Yang dikembalikan berupa *list*, sebab jika terdapat alternatif infleksi *kanji* (salah satu dari `startNewKanji` dan `endNewKanji` tidak *null*) maka *list* tersebut akan berisi 2 jenis infleksi yaitu infleksinya dalam *hiragana* dan infleksinya dalam *kanji*.

Inilah contoh *list* yang dikembalikan berdasarkan contoh-contoh objek pada 3.3.10.3 dan `dictionaryWord` yang diperoleh dari *query*:

- a) Masukan pengguna: “terebi”, `dictionaryWord`: テレビ
Keluaran: {テレビ}
- b) Masukan pengguna: “tabeta”, `dictionaryWord`: 食べる
Keluaran: {食べた}
- c) Masukan pengguna: “daibouken”, ada beberapa kata di basis data dengan bacaan “bouken”:
dengan `dictionaryWord`: 冒険
Keluaran: {だい冒険, 大冒険}
dengan `dictionaryWord`: 剖検
Keluaran: {だい剖検, 大剖検}
dengan `dictionaryWord`: 望見
Keluaran: {だい望見, 大望見}
- d) Masukan pengguna: “omatikudasai”, `dictionaryWord`: 待つ
Keluaran: {お待ちください, 御待ちください}
- e) Masukan pengguna: “kimasita”, `dictionaryWord`: 来る

Keluaran: {来ました}

f) Masukan pengguna: “samuitoit'ta”, dictionaryWord: 寒い

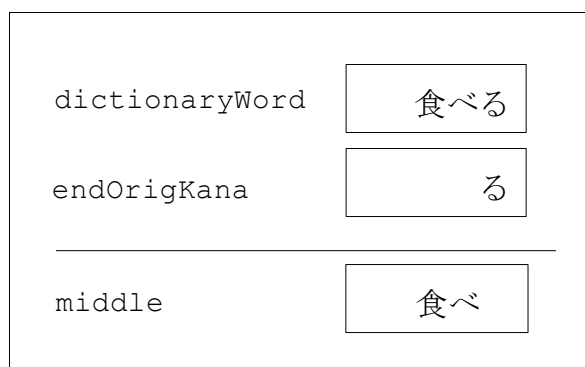
Keluaran: {寒いと行った, 寒いと言った}

Metode ini mengasumsikan argumen yang diberikannya sesuai dengan syarat pada `srcWordKana` dan `srcTag`. Pemberian argumen yang tidak sesuai dapat menghasilkan eksepsi pada operasi *substring* yang dilakukan di dalamnya.

Dengan seluruh data yang ada pada atribut, cara menginfleksikannya sendiri tidaklah susah. Karena infleksi melibatkan penambahan di awal dan penggantian di akhir kata, maka pasti terdapat “tengah” kata yang tidak berubah dari kata dasarnya. Jika katanya ditulis dengan *kanji*, maka tengah itu adalah *kanji* ditambah *okurigana*, jika ada.

Pada metode ini, tengah kata tersebut disimpan di variabel `middle`. Cara mendapatkannya adalah argumen `dictionaryWord` yang dipotong belakangnya sejumlah `panjang(endOrigKana)` (umum) atau sejumlah `panjang(endOrigKana) - 1` (jika `soundMorph` bernilai *true*).

Sebagai contoh, untuk infleksi “taberu” menjadi “tabeta” (contoh b di 3.3.10.3), inilah ilustrasi dari `middle`:



Gambar 3.56 Ilustrasi `middle` untuk `soundMorph false`

Inilah ilustrasi `middle` untuk kasus `soundMorph true`, yaitu contoh e di

3.3.10.3 (“kuru” menjadi “kimasita”):

dictionaryWord	来る
endOrigKana	←る
<hr/>	
middle	来

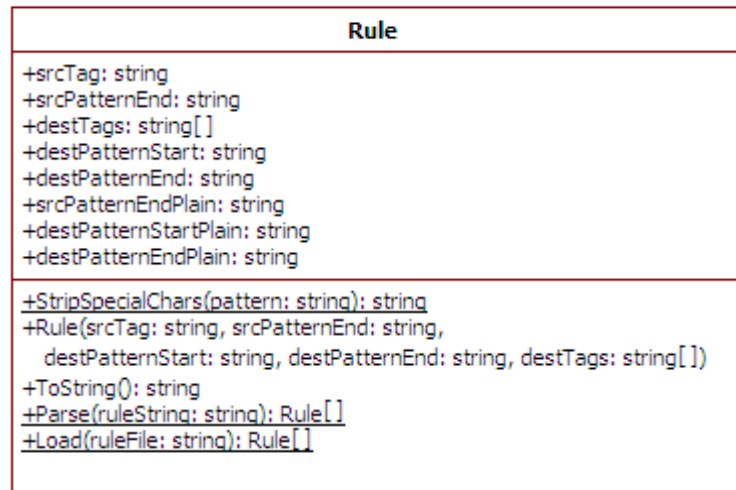
Gambar 3.57 Ilustrasi `middle` untuk `soundMorph true`

Setelah itu, keluarannya adalah *list* yang anggota pertamanya berisi `startNewKana + middle + endNewKana` untuk `soundMorph false`. Untuk `soundMorph true`, `endNewKana` digantikan oleh *substring*-nya yang dimulai dari karakter kedua. Jika terdapat alternatif penulisan kanji untuk infleksi, maka konkatensi *string* yang sama juga akan dilakukan namun dengan variabel `startNewKanji` dan `endNewKanji`.

Inilah ilustrasinya sebagai kelanjutan dari kedua contoh sebelumnya:

3.3.11.2 Perancangan Kelas

Diagram kelas `Rule` adalah sebagai berikut:



Gambar 3.60 Diagram Kelas `Rule`

Secara umum, untuk membuat objek `Rule` digunakan metode `Parse` yang menerima *string* dengan format seperti tertulis pada 3.3.4.4. Terdapat konstruktor, namun dalam `GamaIme.dll` yang menggunakannya hanyalah metode `Parse`.

Untuk menjelaskan mengenai atributnya, akan digunakan contoh-contoh aturan berikut:

- `v5k/*ku/*ita/past`
 - `te/*/*(i|行)ku/v5k-s`
 - `n/*/(dai|大)*/dai`
 - `vs-i/*(su)ru/*(si)ta/past`
- `srcTag` menyimpan penanda kata sumber. Untuk aturan yang menspesifikasikan banyak penanda, misalnya `n,adj-na`, maka metode

Parse akan membuat objek `Rule` untuk masing-masing penanda sumber tersebut. Pada contoh a, nilainya adalah “v5k” dan pada contoh b nilainya adalah “te”.

- `srcPatternEnd` menyimpan akhiran pada pola sumber. Pada contoh a nilainya adalah “ku”, pada contoh b nilainya adalah “”, dan pada contoh d nilainya adalah .
- `destTags` menyimpan penanda tujuan. `srcTag` otomatis terbawa menjadi `destTags`. Pada contoh a nilainya adalah {“v5k”, “past”} dan pada contoh b nilainya adalah {“te”, “v5k-s”}.
- `destPatternStart` menyimpan awalan pada pola tujuan. Pada contoh a nilainya adalah “” dan pada contoh c nilainya adalah “(dai | 大)”.
- `destPatternEnd` menyimpan akhiran pada pola tujuan. Pada contoh a nilainya adalah “ita”, pada contoh b nilainya adalah “(i | 行) ku”, dan pada contoh c nilainya adalah “”.
- `srcPatternEndPlain`, `destPatternStartPlain`, dan `destPatternEndPlain` merupakan masing-masing `srcPatternEnd`, `destPatternStart`, dan `destPatternEnd` yang dihilangkan karakter spesialnya. Yang dimaksud dengan karakter spesial adalah kurung buka, kurung tutup, tanda pipa, dan *kanji*. Jadi yang tersisa hanyalah romajinya. Ini digunakan untuk perbandingan *string* saat perantaraan. Contohnya, “(i | 行) ku” akan menjadi “iku” dan “(su) ru” akan menjadi “suru”. Yang melakukan penghilangan karakter spesial

adalah metode `StripSpecialChars`.

3.3.11.3 Metode `Rule[] Load(string ruleFile)`

Metode ini akan membuka suatu berkas `ruleFile` dan mem-parse aturan-aturan di dalamnya. Format baris aturannya dideskripsikan pada 3.3.4.4. Baris kosong akan dilewati dan baris yang diawali `%` dianggap *comment*. Jika baris berisi perintah khusus “TERMINATE” maka pemrosesan hanya akan dilakukan sampai baris tersebut¹⁶.

Secara internal, metode ini menggunakan metode `Parse` untuk mem-*parse* baris aturan.

Berkas aturan yang digunakan oleh `GamaIme.dll` dinamakan `rules_romaji.txt` dan bisa dilihat di lampiran.

3.3.12 Kelas `RestrictionChecker`

3.3.12.1 Tinjauan

Objek kelas ini bertujuan untuk mengecek apakah suatu kata (*string*) mungkin menjadi anggota dari kelas kata EDICT tertentu. Sebagai contoh, suatu objek `RestrictionChecker` bisa menjawab pertanyaan seperti “*apakah mungkin kata “taberu” merupakan kelas kata EDICT “adj” (keiyoushi atau adjektiva -i)?*” Ini dilakukan dengan memeriksa akhir katanya, sebab banyak¹⁷ kelas kata pada bahasa Jepang yang memiliki akhiran seragam. Sebagai contoh, *keiyoushi* (adjektiva -i) pasti berakhiran “i”.

¹⁶ Fasilitas ini disediakan untuk *debugging*.

¹⁷ Nomina (*meishi*) adalah contoh kelas kata yang tidak memiliki akhiran seragam.

Guna dari pengecekan ini adalah untuk membuang hasil kemungkinan yang jelas-jelas tidak mungkin dalam proses perantaian aturan tata bahasa.

Sebagai contoh, misal pengguna meminta transliterasi dari “atsukunai” (tidak panas). Mesin perantai pertama-tama akan menemukan kemungkinan bahwa “atsukunai” dapat dibentuk dari “atsukuru”, jika terdapat kata “atsukuru” yang kelas katanya $v1$ (verba *ichidan* atau verba -ru). Namun berikutnya `RestrictionChecker` akan menolak kemungkinan tersebut, sebab `RestrictionChecker` tahu bahwa $v1$ pasti berakhiran “iru” atau “eru”, sedangkan “atsukuru” tidak memenuhi syarat tersebut.

Dengan contoh lain yaitu “tabenai” (tidak makan), mesin perantai pertama-tama juga akan menemukan kemungkinan bahwa “tabenai” dapat dibentuk dari “taberu”, jika “taberu” adalah $v1$. Dalam contoh ini, `RestrictionChecker` akan mengatakan bahwa hal tersebut memang mungkin.

Dengan pemangkasan yang dilakukan `RestrictionChecker`, keuntungan yang diperoleh ada dua:

- Kata-kata gadungan tidak akan dirantainya lebih lanjut.
- Tidak akan dilakukan *query* untuk mencari kata-kata gadungan tersebut.

Pengecekan yang dilakukan berbasis *romaji*, bukan *kana*. Terdapat keuntungan dan kekurangan dari pendekatan tersebut. Keuntungannya adalah:

- Kemudahan dalam menspesifikasikan aturan untuk $v1$ (verba *ichidan* atau verba -ru), yaitu hanya diakhiri “iru” atau “eru”. Pendekatan berbasis *kana* mengharuskan dispesifikasikannya banyak akhiran seperti いる “iru”, き

る “kiru”, しる “shiru”¹⁸, ちる “chiru”, dst.

Kekurangannya adalah:

- Tidak bisa memaksa kata yang seharusnya berakhiran dengan *kana* vokal seperti う “u”. Jika diberikan aturan bahwa v5u (verba *godan* atau verba -u baris u) harus berakhiran “u”, maka kata seperti “tobu” dianggap v5u¹⁹. Pada kenyataannya, “tobu” berakhiran kana ぶ “bu” padahal v5u harus berakhiran kana う “u”.

3.3.12.2 restriction.txt

Supaya fleksibel, aturan didefinisikan di berkas teks `restriction.txt`. Berkas ini merupakan berkas eksternal, berdiri sendiri di luar `GamaIme.dll` namun harus berada pada direktori yang sama. Formatnya adalah:

```
tag1[,tag2,...,tagn]:ending1[,ending2,...,endingn]
```

Artinya, untuk setiap penanda yang ada di sebelah kiri “:”, akhiran yang mungkin didaftar di sebelah kanan. Baris kosong diabaikan dan baris yang diawali % dianggap *comment*.

Beberapa contoh:

- `adj:i`

Ini berarti kata dengan penanda `adj` harus diakhiri *string* “i”.

- `v1:iru,eru`

¹⁸ Pada EDICT sebetulnya tidak ada verba *ichidan* yang diakhiri “shiru”.

¹⁹ Contohnya adalah saat pengguna meminta transliterasi “tobi”, karena mesin perantai akan menemukan aturan yang menyatakan bahwa “tobi” bisa dibentuk dari v5u “tobi” (bandingkan dengan 会い “ai” yang memang dibentuk dari v5u 会う “au”).

Ini berarti kata dengan penanda `v1` harus diakhiri *string* “iru” maupun “eru”.

- `v5u,v5u-s:u`

Ini berarti kata dengan penanda `v5u` dan kata dengan penanda `v5u-s` harus diakhiri string “u”. Baris tersebut ekuivalen dengan dua baris berikut:

```
v5u:u
v5u-s:u
```

Inilah isi dari `restriction.txt`:

```
% format:
% tag1[,tag2,...,tagN]:ending1[,ending2,...,endingN]

% example:
% vs-i,vs-s:suru
% the restriction above means that all vs-i and vs-s must
end with suru

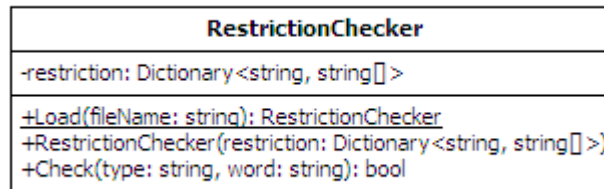
adj:i
v1:iru,eru
v5s:su
v5k:ku
v5k-s:iku,yuku
v5g:gu
v5b:bu
v5t:tu
v5m:mu
v5r:ru
v5aru:aru
v5uru:uru
v5n:nu
v5u,v5u-s:u
vs-i,vs-s:suru
vz:zuru
vk:kuru

te:te,de
```

Gambar 3.61 `restriction.txt`

3.3.12.3 Perancangan Kelas

Inilah diagram kelas dari `RestrictionChecker`:



Gambar 3.62 Diagram Kelas `RestrictionChecker`

Aturan direpresentasikan secara internal menggunakan `Dictionary`. Kuncinya berupa kelas kata dan nilai dari kunci tersebut adalah daftar akhiran yang mungkin. Aturan bisa dibuat sendiri secara programatis lalu diserahkan ke konstruktornya. Namun pada umumnya yang digunakan adalah metode statik `Load` yang akan mem-*parse* berkas teks dengan format yang telah dijelaskan pada 3.3.12.2. Untuk mengecek apakah suatu kata memenuhi batasan suatu kelas kata, digunakan metode `Check`. Metode `Check` akan mengembalikan `true` jika argumen `type` tidak terdapat pada aturan.

Yang cukup menarik adalah penggunaan `Regex` pada metode `Load` jadi hanya metode tersebut yang akan dibahas lebih lanjut cara kerja internalnya.

3.3.12.4 Metode Statik void `Load(string fileName)`

Untuk mengecek apakah suatu string sesuai dengan format baris pada 3.3.12.2 digunakan ekspresi reguler. Ekspresinya adalah:

```
^(?<tags>[a-zA-Z0-9-]+(,[a-zA-Z0-9-]+)*) : (?<endings>[a-z ']+(,[a-z ']+)*)$
```

Penjelasannya adalah:

- `^` berarti awal *string*, dan `$` berarti akhir *string*
- `(?<tags>[a-zA-Z0-9-]+(,[a-zA-Z0-9-]+)*)` di sebelah kiri “:” adalah pola untuk daftar penanda. Grup tersebut diberi nama `tags`. Tiap penanda polanya adalah `[a-zA-Z0-9-]+` yang berarti satu atau lebih karakter alfanumeris atau “-”. Contoh penanda yang cocok dengan pola tersebut adalah “adj” dan “adj-na”.
- Titik dua adalah pembatas antara daftar penanda dengan daftar akhiran.
- `(?<endings>[a-z']+(,[a-z']+)*)` di sebelah kanan “:” adalah pola untuk daftar akhiran yang mungkin. Grup tersebut diberi nama `endings`. Pola untuk tiap penanda adalah satu atau lebih karakter alfanumeris atau apostrof. Guna apostrof adalah untuk mengantisipasi penggunaan silabel *h* (ditransliterasi sebagai *n'*) dan *tsu* (“tsu” kecil, ditransliterasi sebagai *t'* pada mesin perantai).

Untuk melakukan pencocokan pola, pertama dibuat objek `Regex` yang didefinisikan pada namespace `System.Text.RegularExpressions`:

```
Regex restrictionFormat = new Regex(@"^(?<tags>[a-zA-Z0-9-]+(,[a-zA-Z0-9-]+)*) : (?<endings>[a-z']+(,[a-z']+)*) $");
```

Pencocokan dilakukan dengan metode `Match`. Bila cocok maka properti `Success` dari objek `Regex` akan bernilai `true`:

```
Match match = restrictionFormat.Match(line);
```

Dengan pemberian nama pada grup `tags` dan `endings`, *string*-nya bisa diperoleh dengan mudah menggunakan pengindeks `Groups` seperti berikut:

```
string[] endings =
match.Groups["endings"].Value.Split(',');
```

3.3.12.5 Contoh Penggunaan

Inilah contoh penggunaan kelas `RestrictionChecker`, menggunakan “`restriction.txt`” seperti pada 3.3.12.2.

```
RestrictionChecker checker =
RestrictionChecker.Load("restriction.txt");
Console.WriteLine(checker.Check("adj", "kantan")); // false
Console.WriteLine(checker.Check("v1", "magaru")); // false
Console.WriteLine(checker.Check("v5b", "tobu")); // true
Console.WriteLine(checker.Check("v5u", "tobu")); // true
Console.WriteLine(checker.Check("asal", "muri")); // true
```

Gambar 3.63 Contoh Penggunaan `RestrictionChecker`

3.3.13 Kelas `KanjiTransliterator`

3.3.13.1 Tinjauan

`KanjiTransliterator` adalah kelas yang melakukan transliterasi dari *romaji* ke *kanji*, inti dari `GamaIme.dll`. Algoritmanya dijelaskan pada 3.3.4.

3.3.13.2 Perancangan Kelas

Inilah diagram kelas dari `KanjiTransliterator`:

KanjiTransliterator
<pre> +kanaTransliterator: RomajiToKanaTransliterator = new RomajiToKanaTransliterator() -restrictionChecker: RestrictionChecker -rules: Rule[] -edictTags: string[] -dbFileName: string </pre>
<pre> +KanjiTransliterator(ruleFileName: string, restrictionFileName: string, tagFileName: string, dbFileName: string) +Transliterate(inputRomaji: string): string[] +FetchDb(instancesByLevel: List<List<RuleInstance>>): string[] +HiraganaToKatakana(input: string): string -AddDbResult(popResult: Dictionary<int, List<string>>, trans: string, pop: int): void +GetNextInstances(instance: RuleInstance): List<RuleInstance> -GetKanjiPattern(startPattern: string): string </pre>

Gambar 3.64 Diagram Kelas KanjiTransliterator

Penjelasan dari atributnya adalah:

- `kanaTransliterator` digunakan untuk mentransliterasi *romaji* ke *kana*. Objek ini digunakan saat perantaraan untuk mengisi nilai-nilai `RuleInstance` yang sedang dibuat (misalnya pada Gambar 3.33).
- `restrictionChecker` digunakan untuk mengecek apakah suatu kata memenuhi aturan fonologis yang ditentukan oleh kelas katanya. Objek ini digunakan untuk membuang kata-kata gadungan yang ditemukan saat perantaraan (Gambar 3.30).
- `rules` adalah larik seluruh aturan bahasa yang ada.
- `edictTags` adalah larik yang berisi penanda EDICT yang mungkin. Larik ini digunakan untuk mengecek apakah suatu `RuleInstance` adalah `RuleInstance` perantara atau bukan (3.3.4.7 *pseudocode* 3.a.ii). Berkas yang berisi penanda-penanda tersebut ada di `edict_tags.txt`.
- `dbFileName` adalah nama berkas SQLite yang akan dimuat saat

melakukan query kata dasar.

Diagram yang mengilustrasikan hubungan kelas ini dengan kelas-kelas lainnya pada program ada di lampiran A.

3.3.13.3 Contoh Penggunaan

Inilah contoh penggunaan kelas KanjiTransliterator:

```
KanjiTransliterator t =
    new KanjiTransliterator("rules_romaji.txt",
        "restriction.txt", "edict_tags.txt", "edict.db3");
foreach(string s in t.Transliterate("mot'teikimasita"))
{
    Console.WriteLine(s);
}
```

Gambar 3.65 Contoh Penggunaan Kelas KanjiTransliterator

3.3.13.4 Konstruktor KanjiTransliterator(string ruleFileName, string restrictionFileName, string tagFileName, string dbFileName)

Konstruktor ini menginisialisasi suatu KanjiTransliterator agar siap digunakan untuk transliterasi. Argumennya adalah:

1. `ruleFileName`, berkas aturan tata bahasa dengan format seperti yang dijelaskan pada 3.3.4.4.
2. `restrictionFileName`, berkas untuk menginisialisasi `RestrictionChecker`, dengan format dijelaskan pada 3.3.12.2. Penggunaannya dalam perantaraan ada di Gambar 3.30.
3. `tagFileName`, berkas yang berisi daftar penanda-penanda EDICT yang dibolehkan. Gunananya adalah untuk mengecek apakah suatu `RuleInstance` merupakan bentuk perantara atau bukan (lihat 3.3.4.5

dan 3.3.4.7 pseudocode 3.a.ii). Formatnya sama seperti pada 3.2.4.5 dengan catatan bahwa harus ada penanda yang bernama `any` agar kata-kata dengan penanda tersebut dicari di basis data.

4. `dbFileName`, berkas basis data yang dihasilkan oleh `EdictDbBuilder.exe` (3.2).

3.3.13.5 Metode `string[] KanjiTransliterator.Transliterate(string inputRomaji)`

Metode inilah yang melakukan transliterasi dari *romaji* ke *kanji*. Pertama-tama dilakukan pencarian seluruh `RuleInstance` yang bisa menghasilkan `inputRomaji` sebagaimana digambarkan pada 3.3.4.7. Perbedaannya adalah bagaimana `RuleInstance`-nya diorganisir.

`RuleInstance`-nya diorganisir dalam *List* dari *List* `RuleInstance`. Anggota pertama berisi *List* dari `RuleInstance` yang berada di tingkat tertinggi pohon pencarian. Anggota kedua berisi *List* dari `RuleInstance` yang berada di tingkat bawahnya, dan seterusnya. Ini digunakan untuk mensortir hasil transliterasi.

3.3.13.6 Metode `string[] FetchDb(List<List<RuleInstance>> instancesByLevel)`

Metode ini akan dipanggil oleh `Transliterate` untuk melakukan query ke basis data, infleksi berdasar `RuleInstance`, dan mensortir hasilnya berdasarkan prioritas berikut:

1. Posisinya pada pohon pencarian
2. Tingkat popularitasnya yang terdapat di basis data
3. Urutan kemunculannya pada *loop*
4. Penulisan *hiragana* untuk entri basis data yang memiliki penanda uk akan didahulukan daripada penulisan *kanji*-nya.
5. Hasil infleksi *hiragana* akan didahulukan daripada hasil infleksi *kanji*-nya.

FetchDb secara otomatis akan menambahkan transliterasi hiragana dan katakana pada akhir.

3.4 Komponen Perantara Klien Server: transliterator.aspx

`transliterator.aspx` merupakan suatu aplikasi ASP.NET kecil yang menerima *request* HTTP AJAX dari klien dan, menggunakan `GamaIme.dll`, mengembalikan transliterasinya.

Masukan bagi `transliterator.aspx` adalah:

- Kata yang ingin ditransliterasi dalam *hiragana*. Kata ini harus diletakkan di variabel HTTP GET `hiragana`.

Keluarannya adalah:

- Daftar trasliterasinya yang dipisah dengan titik koma.

Inilah contoh cara memanggil `transliterator.aspx`:

```
http://server.com/transliterator.aspx?hiragana=たべなかつた
```

Dan inilah dokumen yang akan dihasilkan:

```
食べなかつた;たべなかつた;タバナカッタ
```

Untuk mentransliterasi, digunakan metode

KanjiTransliterator.Transliterate. Karena metode
 Trasliterate membutuhkan masukan dalam *romaji* GamaIme.dll, maka
 masukan *hiragana* dari pengguna akan sebelumnya ditransliterasi dulu menjadi
romaji menggunakan metode
 HiraganaToRomajiTransliterator.Transliterate.

3.5 Komponen Aplikasi Web IME

3.5.1 Tinjauan

Aplikasi web IME merupakan antarmuka bagi pengguna untuk menuliskan teks bahasa Jepang. Pada aplikasi web ini terdapat suatu kotak teks tempat pengguna bisa mengetik. Jika tombol IME diaktifkan, maka masukan dari pengguna akan ditangani oleh IME sebagaimana cara kerja IME yang terinstall di sistem operasi. Setelah pengguna mengetik di kotak teks tersebut, teksnya bisa di-*copy* dan *paste* ke aplikasi lain.

3.5.2 Analisis Permasalahan

IME perlu mengubah masukan dari pengguna menjadi teks bahasa Jepang, lalu menyisipkannya di kotak teks. Elemen HTML `textarea` tidak mendukung fungsionalitas untuk menyisipkan teks maupun *query* posisi kursor sehingga tidak bisa digunakan. Cara yang paling mudah adalah menggunakan `iframe` yang berisi halaman kosong²⁰. Properti `designMode` pada dokumen `iframe` tersebut

²⁰ Alternatif lain adalah mengembangkan suatu `textarea` sendiri yang memiliki fitur untuk menyisipkan teks pada kursor. Alternatif tersebut telah dicoba dengan menangani seluruh *event* pada elemen `div`, namun ditinggalkan karena di tengah pengembangan terlihat jelas bahwa alternatif tersebut akan memerlukan waktu pengembangan yang sangat banyak.

lalu bisa diset menjadi 'On' sehingga fungsionalitas pengetikan menjadi aktif pada `iframe` tersebut, ditambah lagi tersedia fungsi untuk menyisipkan sembarang kode HTML ke posisi kursor.

IME perlu banyak berurusan dengan masukan *keyboard* dari pengguna. Karena terdapat perbedaan signifikan dalam penanganan *event keyboard* di Firefox, IE, dan Opera, maka dibuat suatu *layer* yang menangani perbedaan tersebut sehingga menyederhanakan kode yang menggunakannya.

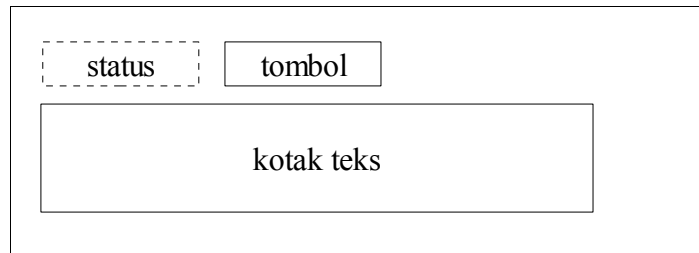
Untuk meminta transliterasi ke server, digunakan teknologi AJAX. Namun karena sangat mungkin bahwa pengguna meminta transliterasi yang sama berulang kali, maka dibuat suatu sistem *caching* agar menghemat waktu.

3.5.3 Analisis Kebutuhan Sistem

Semua kebutuhan yang tertera pada 3.5.2 dapat ditangani oleh *browser* modern. Secara spesifik, pengembangan aplikasi web IME ini ditargetkan pada *browser* Firefox 2, Opera 9, dan Internet Explorer 6.

3.5.4 Perancangan Antarmuka

Antarmuka utamanya sangatlah sederhana, yaitu kotak teks untuk mengetik, tombol untuk menyalakan/mematikan IME, dan status IME (menyala atau tidak):



Gambar 3.66 Perancangan Antarmuka Utama Aplikasi Web IME

Jika IME aktif, maka masukan pengguna akan ditangkap dan tidak langsung dimasukkan ke kotak teks, namun ditampilkan pada *popup* IME. Pada *popup* tersebut pengguna bisa melihat/memodifikasi kata yang telah dia masukkan dan memilih alternatif transliterasi yang ada.



Gambar 3.67 Perancangan Antarmuka Popup IME

Pada Gambar 3.67, kotak *loading* hanya akan ditampilkan saat menunggu respons AJAX dan kotak pilihan transliterasi hanya akan ditampilkan saat pengguna melakukan pemilihan alternatif transliterasi.

Popup IME memiliki beberapa keadaan yaitu:

0. hidden: IME sedang tidak aktif atau IME aktif namun pengguna belum memberi masukan. *Popup* tidak ditampilkan.
1. typing: Pengguna sedang mengetikkan masukan. Hanya kotak masukan yang ditampilkan.
2. selected: Pengguna memilih konversi paksa (misalnya konversi paksa ke katakana lebar penuh atau katakana lebar setengah). Konversi paksa bisa dilakukan dengan tombol khusus seperti F7 untuk konversi paksa ke katakana lebar penuh. Hanya kotak masukan yang ditunjukkan.
3. selecting: Pengguna sedang memilih dari daftar transliterasi yang mungkin. Kotak pilihan transliterasi dapat ditunjukkan maupun disembunyikan.
4. loading: IME sedang menunggu respons AJAX. Kotak *loading* akan ditampilkan setelah selang waktu tertentu.

Setiap keadaan dapat berpindah ke keadaan lain karena kejadian tertentu.

Inilah tabel perubahan keadaannya:

Tabel 3.3 Tabel Perubahan Keadaan *Popup* IME

Keadaan Lama	Kejadian	Keadaan Baru
0 (hidden)	masukan diketikkan	1 (typing)
1 (typing)	ESC ditekan, <i>backspace</i> ditekan sehingga karakter terakhir terhapus, daerah di luar <i>popup</i> diklik	0 (hidden)
	tombol konversi paksa (misal F7) ditekan	2 (selected)
	spasi ditekan (masukan ada di cache)	3 (selecting)
	spasi ditekan (masukan tidak ada di cache)	4 (loading)
2 (selected)	masukan baru diketikkan, ESC atau <i>backspace</i>	1 (typing)

	ditekan	
	spasi ditekan (masukan ada di cache)	3 (selecting)
	spasi ditekan (masukan tidak ada di cache)	4 (loading)
3 (selecting)	masukan baru diketikkan, ESC atau <i>backspace</i> ditekan	1 (typing)
	tombol konversi paksa (misal F7) ditekan	2 (selected)
4 (loading)	ESC atau <i>backspace</i> ditekan	1 (typing)
	tombol konversi paksa (misal F7) ditekan	2 (selected)
	repons AJAX diterima	3 (selecting)

Tombol konversi paksa yang ada adalah sebagai berikut:

Tabel 3.4 Tombol Konversi Paksa IME

Tombol	Hasil Konversi	Contoh
F6	<i>hiragana</i>	わたし
F7	<i>katakana</i> lebar penuh	ワタシ
F8	<i>katakana</i> lebar setengah	ワタシ
F9	<i>romaji</i> lebar penuh	w a t a s h i
F10	<i>romaji</i> lebar setengah	watashi

3.5.5 Penanganan Keyboard

Firefox, IE, dan Opera memiliki perilaku yang berbeda jika suatu tombol ditekan terus menerus.

- a) Untuk tombol yang berkorespondensi dengan karakter yang bisa diprint seperti “a”:

1. **Firefox:** Berurutan ditembakkan *event* *keydown* yang diikuti *keypress* (*keydown*, *keypress*, *keydown*, *keypress*, ...). *Event* *keydown* menyimpan informasi di *keyCode* sedang pada *keypress* *keyCode* selalu bernilai 0.

2. **IE:** Berurutan ditembakkan *event* `keydown` yang diikuti `keypress` (`keydown`, `keypress`, `keydown`, `keypress`, ...). Keduanya menyimpan informasi *event* di `keyCode` namun nilainya dapat berbeda. Sebagai contoh, menekan tombol 'a' tanpa *Caps Lock* aktif dan tanpa menekan *shift* akan menembakkan event `keydown` dengan `keyCode` 65 (ASCII 'A') yang diikuti `keypress` dengan `keyCode` 97 (ASCII 'a').
 3. **Opera:** Ditembakkan `keydown` sekali yang diikuti `keypress` berturut-turut (`keydown`, `keypress`, `keypress`, `keypress`, ...). Keduanya menyimpan informasi *event* di `keyCode` namun nilainya dapat berbeda sebagaimana pada kasus IE.
- b) Untuk tombol spesial seperti "shift":
1. **Firefox:** Berurutan ditembakkan hanya *event* `keydown` yang menyimpan informasi di `keyCode` (`keydown`, `keydown`, `keydown`, `keydown`, ...).
 2. **IE:** Berurutan ditembakkan hanya *event* `keydown` yang menyimpan informasi di `keyCode` (`keydown`, `keydown`, `keydown`, `keydown`, ...).
 3. **Opera:** Ditembakkan `keydown` sekali yang diikuti `keypress` berturut-turut (`keydown`, `keypress`, `keypress`, `keypress`, ...). Keduanya menyimpan informasi *event* di `keyCode`.

Berdasarkan kasus b1 dan b2, hanya `keydown` yang bisa digunakan untuk

mendeteksi penekanan tombol dengan akurat. Namun untuk aksi yang menangani terjadinya penekanan itu sendiri, di Firefox dan IE `keydown` harus digunakan sedangkan di Opera `keypress` harus digunakan.

3.5.6 Meminta Transliterasi ke Server dengan AJAX

Transliterasi yang disediakan server diminta melalui permintaan HTTP GET dengan AJAX. Karena pengguna bisa menggagalkan permintaan sebelum responsnya diterima, tiap permintaan diberi suatu *id* yang unik.

Saat respons diterima dengan baik, hasilnya diletakkan di *cache*. Jika ternyata pengguna masih menunggu respons ini, maka alternatif transliterasi pertama akan dipilih.

Jika respons gagal diperoleh, maka akan dilakukan permintaan ulang setelah *delay* beberapa saat.

3.5.7 Implementasi

Berkas utama IME adalah `ime.htm`, yang mereferensi berkas JavaScript `events.js`, `misc.js`, `keyboard.js`, `transliterator.js`, dan `ime.js`. Selain itu terdapat `blank.htm` yang merupakan halaman kosong untuk kotak teks dan `popup.css` yang merupakan style sheet untuk popup IME. Penjelasannya terdapat pada bagian-bagian selanjutnya.

3.5.8 `misc.js`

Pada berkas ini didefinisikan fungsi-fungsi yang mempermudah

penjelajahan dan manipulasi DOM. Sebagai contoh, pada potongan HTML berikut:

```
<div>
  <p id="foo">Paragraf 1</p>
  <p>Paragraf 2</p>
</div>
```

Maka skrip berikut:

```
document.getElementById("foo").nextSibling
```

Tidaklah mengembalikan paragraf kedua, namun elemen DOM teks yang berisi *whitespace*. Ini karena dalam penulisannya, `<p>` tidak langsung muncul mengikuti `</p>` tetapi diselingi baris baru dan tab.

Normalnya, elemen-elemen teks tersebut tidaklah dibutuhkan. Oleh karenanya, terdapat fungsi yang mengabaikan elemen-elemen tersebut. Contohnya adalah fungsi `next` berikut:

```
function next(elem)
{
  do
  {
    elem = elem.nextSibling;
  }while(elem && elem.nodeType != 1);
  return elem;
}
```

Gambar 3.68 Fungsi `next` yang mengabaikan elemen teks

Dengan fungsi `next` tersebut, skrip berikut:

```
next(document.getElementById("foo"))
```

Akan mengembalikan paragraf kedua.

Fungsionalitas dari masing-masing fungsi pada `misc.js` diberikan pada *comment* sebelum fungsi tersebut, yang bisa dilihat pada lampiran.

3.5.9 transliterator.js

Di dalam berkas ini didefinisikan kelas `Transliterator` yang berfungsi melakukan transliterasi *string* linier *greedy* dengan prinsip yang sama seperti pada 3.3.5. Perbedaannya implementasinya adalah:

- Kelas `Transliterator` hanya mendukung transliterasi *string* ke *string*, tidak transliterasi umum dari tipe sumber *S* ke tipe tujuan *D*.
- Format *string* penambahan aturan beda. Kelas `Transliterator` pada `transliterator.js` memiliki format berikut:

```
[sumber_1] [tujuan_1];...;[sumber_n] [tujuan_n]
```

Karakter dapat di-*escape* dengan menggunakan “?”. Sebagai contoh, “?” akan di-*escape* sebagai “;”.

Inilah contoh penggunaannya:

```
var trans = new Transliterator();
trans.addRule("a 1;b 2;c 3;ab 0");
var result = trans.transliterate("bca-abc"); // 231-03
```

Pada `transliterator.js`, terdapat objek `Transliterator` berikut:

- `hiraganaTrans`, yang mentransliterasi *romaji* ke hiragana, lalu simbol dan angka ke karakter lebar penuh.
- `katakanaTrans`, yang mentransliterasi *romaji* ke katakana, lalu simbol dan angka ke karakter lebar penuh.
- `halfwidthTrans`, yang mentransliterasi *romaji* ke katakana lebar setengah.
- `fullwidthTrans`, yang mentransliterasi huruf latin, simbol, dan angka

ke karakter lebar penuh.

3.5.10 keyboard.js

Pada Firefox, IE, dan Opera, terdapat variabel `event.keyCode` pada event-event yang berhubungan dengan *keyboard*, misalnya `keydown`. Dari nilai `event.keyCode`, bisa diketahui tombol yang ditekan pada *keyboard*.

Masalahnya adalah tidak terdapat konsistensi kode antar *browser*. Sebagai contoh, di Firefox dan IE tombol yang berkorespondensi dengan karakter ` dan ~ (tombol di sebelah kiri angka 1) memiliki kode 192, sedangkan di Opera kodenya adalah 96.

`keyboard.js` bertujuan untuk memfasilitasi abstraksi kode yang berbeda-beda pada tiap *browser*. Pada `keyboard.js`, terdapat tabel kode karakter keyboard untuk Firefox, IE, dan Opera. Objek-objeknya adalah sebagai berikut:

Tabel 3.5 Objek Tabel Kode Karakter Keyboard pada `keyboard.js`

Nama Objek	Deskripsi
<code>charTable.special.firefox</code>	Tabel kode karakter spesial (misal shift) untuk Firefox
<code>charTable.special.opera</code>	Tabel kode karakter spesial (misal shift) untuk Opera
<code>charTable.special.explorer</code>	Tabel kode karakter spesial (misal shift) untuk IE
<code>charTable.printable.firefox</code>	Tabel kode karakter normal (misal 'a') untuk Firefox
<code>charTable.printable.opera</code>	Tabel kode karakter normal (misal 'a') untuk Opera
<code>charTable.printable.explorer</code>	Tabel kode karakter normal (misal 'a') untuk IE

Tabel karakter spesial memetakan kode ke nama tombol tersebut. Sebagai

contoh, `charTable.special.firefox[27]` bernilai “esc” dan `charTable.special.firefox[16]` bernilai “shift”.

Tabel karakter normal memetakan kode ke suatu *string* yang panjangnya satu atau dua. Jika panjangnya dua, karakter pertama adalah karakter yang diharapkan muncul pada *widget* masukan jika *shift* tidak ditekan, sedangkan karakter kedua adalah karakter yang diharapkan muncul jika *shift* ditekan. Sebagai contoh, `charTable.printable.firefox[53]` bernilai “5%”.

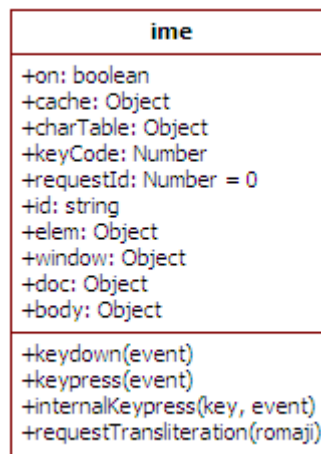
Kode untuk huruf latin tidak terdapat di tabel karena nilainya berurutan sehingga bisa dihitung dengan mudah. Tabel ini digunakan sebagai dasar bagi kode abstraksi *keyboard* pada `ime.js`.

3.5.11 ime.js

Pada berkas ini didefinisikan dua objek penting yaitu `ime` yang merepresentasikan `iframe` kotak teks IME dan `imePopup` yang merepresentasikan *popup* IME.

3.5.11.1 Diagram Kelas dan Atribut ime

Inilah diagram kelas dari `ime`:



Gambar 3.69 Diagram Kelas ime

Penjelasan dari atributnya:

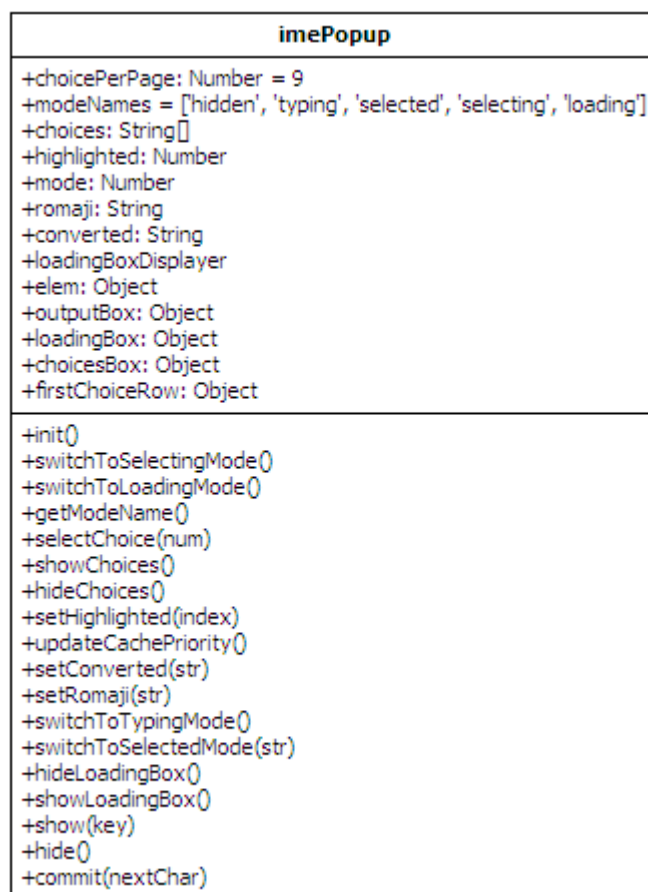
- `on` menunjukkan apakah IME sedang aktif atau tidak
- `cache` menyimpan transliterasi yang sudah pernah diminta dari server. `cache` merupakan *Dictionary* dari *string romaji* ke larik *string* transliterasi.
- `charTable` berisi tabel karakter sesuai dengan browser yang digunakan sebagaimana dijelaskan di 3.5.10. `charTable` memiliki member `printable` dan `special`.
- `keyCode` merupakan *integer* yang berisi kode dari tombol keyboard terakhir yang ditekan.
- `requestId` adalah *id* dari permintaan AJAX yang terakhir. Variabel ini digunakan untuk melacak apakah suatu respons AJAX memang masih diinginkan oleh pengguna.
- `id` merupakan *string* yang berisi id HTML dari `iframe` IME.

- elem berisi referensi ke elemen DOM `iframe` IME.
- window berisi referensi ke window dari `iframe` IME.
- doc berisi referensi ke elemen DOM `document` dari `iframe` IME.
- body berisi referensi ke elemen DOM `body` dari `iframe` IME.

Diagram yang mengilustrasikan hubungan kelas ini dengan kelas-kelas lainnya pada program ada di lampiran A.

3.5.11.2 Diagram Kelas dan Atribut `imePopup`

Inilah diagram kelas `imePopup`:



Gambar 3.70 Diagram Kelas `imePopup`

Penjelasan dari atributnya:

- `choicePerPage` adalah konstan yang menunjukkan jumlah pilihan dalam suatu halaman `imePopup`.
- `modeNames` adalah konstan yang menunjukkan nama-nama mode `imePopup`. Konstan ini digunakan oleh metode `getModeName`.
- `choices` merupakan pilihan transliterasi yang tersedia.
- `highlighted` menunjukkan index transliterasi yang dipilih dari `choices`.
- `mode` menunjukkan keadaan `imePopup` (misal 0 berarti `hidden`).
- `romaji` menunjukkan masukan pengguna.
- `converted` menunjukkan *string* hasil transliterasi.
- `loadingBoxDisplayer` adalah *timer* yang menampilkan kotak *loading* jika respons AJAX tidak diterima dalam jangka waktu tertentu.
- `elem` berisi referensi ke elemen `div imePopup` pada DOM.
- `outputBox` berisi referensi ke elemen DOM untuk kotak keluaran (hasil transliterasi) pada `imePopup`.
- `loadingBox` berisi referensi ke elemen DOM untuk kotak loading pada `imePopup`.
- `choicesBox` berisi referensi ke elemen DOM untuk kotak pilihan pada `imePopup`.
- `firstChoiceRow` berisi referensi ke elemen DOM untuk baris pertama pada kotak pilihan pada `imePopup`.

Diagram yang mengilustrasikan hubungan kelas ini dengan kelas-kelas lainnya pada program ada di lampiran A.

3.5.11.3 Inisialisasi ime

Objek IME dituliskan ke halaman HTML dengan memanggil fungsi `writeIme`. Inilah kode yang menuliskannya:

```
document.writeln('IME is <span id="imeStatus">off</span>.  
<input id="imeButton" type="button" value="Turn IME On"  
onclick="switchIme()"></input><br /><br />');  
document.writeln('<iframe id="' + ime.id + '" name="' +  
ime.id + '" src="blank.htm" style="width: ' + width + 'px;  
height: ' + height + 'px;"></iframe>');
```

Gambar 3.71 Kode `writeIme` yang Menuliskan IME-nya

Setelah IME dituliskan ke HTML, fungsi `writeIme` juga menginisialisasi atribut-atribut pada objek `ime`, mengaktifkan mode `designMode`, dan memasang penanganan-penanganan *event* yang bersesuaian.

`imePopup` juga diinisialisasi di fungsi `writeIme`. Inilah kode HTML dari *popup* IME tersebut:

```
<div id="popup" style="position: absolute;">  
<div><span id="popupConverted" style="background: white;  
border-bottom: dashed black 1px;">わたし</span></div>  
<div id="popupLoading" style="display: none"><span  
style="background: rgb(220,220,220); color: black; padding-  
left: 5px; padding-right: 5px; border: solid black  
3px;">Loading...</span></div>  
<table id="popupTable" class="popupTable" cellspacing="0"  
style="background: rgb(220,220,220); border: solid black 3px;  
position: relative; left: -22px; display: none;">  
<tr id="popupTableFirstRow"><td class="popupNumber">1</td><td  
class="popupChoice">bla</td></tr>  
<tr><td class="popupNumber">2</td><td class="popupChoice">bla  
bla bla</td></tr>  
<tr><td class="popupNumber">3</td><td  
class="popupChoice">bla</td></tr>  
<tr><td class="popupNumber">4</td><td  
class="popupChoice">bla</td></tr>
```

```

<tr><td class="popupNumber">5</td><td
class="popupChoiceSelected">bla</td></tr>
<tr><td class="popupNumber">6</td><td
class="popupChoice">bla</td></tr>
<tr><td class="popupNumberDisabled">7</td><td
class="popupChoiceDisabled"></td></tr>
<tr><td class="popupNumberDisabled">8</td><td
class="popupChoicDisablede"></td></tr>
<tr><td class="popupNumberDisabled">9</td><td
class="popupChoiceDisabled"></td></tr>
<tr><td colspan="2" style="border-top: ridge; text-align:
right">3/7</td></tr>
</table>
</div>

```

Gambar 3.72 Kode HTML Popup IME

Fungsi `writeIme` juga berikutnya akan memasang penanganan-penanganan event yang bersesuaian untuk `popupIme`.

3.5.11.4 Penanganan Keyboard

Penanganan keyboard dibagi pada 3 fungsi yaitu `ime.keydown`, `ime.keypress`, dan `ime.internalKeypress`.

`ime.keydown` akan menangani *event* `keydown` pada browser. Hanya pada *event* tersebutlah informasi akurat mengenai tombol *keyboard* yang ditekan bisa diperoleh. Pada metode ini nilai `event.keyCode` disimpan di `ime.keyCode`.

`ime.keypress` akan menangani *event* `keydown` pada Firefox dan IE, sedangkan pada Opera akan menangani *event* `keypress`. Dipanggilnya metode ini berarti telah terjadi penekanan *keyboard* yang juga mendukung penekanan berturut-turut.

Pada metode `ime.keypress`, kode dari `keyCode` akan diterjemahkan menjadi bentuk yang lebih mudah dipahami pemrogram. Untuk keperluan ini

digunakan tabel pada `ime.charTable`.

Bagi tombol keyboard yang bisa dicetak di kotak teks, penerjemahannya adalah karakter *unicode* dari tombol tersebut. Sebagai contoh, hasil terjemahan tombol 'a' pada *keyboard* adalah string 'a' sendiri.

Bagi tombol spesial, penerjemahannya adalah representasi tekstual dari tombol tersebut yang panjangnya lebih dari 1 karakter. Sebagai contoh, hasil terjemahan tombol *enter* pada *keyboard* adalah *string* 'enter'.

Setelah proses penerjemahan tersebut, dipanggilah metode `ime.internalKeyPress`. Metode tersebut sudah tidak perlu lagi repot berurusan dengan `keyCode` karena konversinya ke bentuk yang lebih mudah dipahami sudah tersedia. Dengan begitu, kode pada `ime.internalKeyPress` lebih mudah dibaca.

3.5.11.5 Permintaan Transliterasi ke Server dan Caching

Transliterasi pada server diminta dengan suatu request AJAX melalui metode `ime.requestTransliteration`. Metode ini akan menambahkan nilai `ime.requestId` dan menyimpannya.

Jika respons AJAX telah diterima, maka pertama dicek apakah transliterasinya sudah ada di *cache*. Jika ya, berarti respons ini datang terlalu lambat dan diabaikan. Jika tidak, maka pilihan transliterasinya disimpan di *cache* sehingga jika selanjutnya pengguna meminta transliterasi yang sama, tidak perlu dilakukan pemanggilan AJAX.

```
if(typeof ime.cache[romaji] == 'undefined')
```

```
{
    ime.cache[romaji] = xml.responseText.split(';');
}
```

Gambar 3.73 Pengecekan Cache setelah Respons AJAX diterima

Selanjutnya dicek apakah pengguna masih menunggu transliterasinya ataukah sudah berganti pikiran²¹. Ini dilakukan dengan mengecek apakah `ime.requestId` masih sama dengan `id` yang disimpan pada fungsi (berarti ini adalah request yang terakhir) dan apakah saat ini `imePopup` masih berada pada mode `loading` (berarti pengguna sedang menunggu transliterasi). Jika ya, maka IME dipindah ke mode `selecting`:

```
if(requestId == ime.requestId && imePopup.getModeName() ==
'loading')
{
    imePopup.switchToSelectingMode();
}
```

Gambar 3.74 Perpindahan ke Mode `selecting` setelah Respons AJAX diterima

3.5.11.6 Fungsi `imePopup.init`

Fungsi ini dipanggil oleh fungsi `writelme` setelah elemen DOM dari `imePopup` dibuat. Yang dilakukan fungsi ini adalah menginisialisasi atribut-atribut pada `imePopup`.

3.5.11.7 Fungsi `imePopup.setHighlighted(index: Number)`

Fungsi digunakan untuk memilih pilihan tertentu pada kotak pilihan transliterasi.

Fungsi ini menerima masukan `index` berupa bilangan bulat. Bilangan

²¹ Misalnya saat *loading*, pengguna sadar bahwa kata yang diketikkannya salah sehingga memperbaikinya.

tersebut akan dibungkus ke jangkauan 0 sampai `imePopup.choices.length - 1`, dalam artian bahwa jika nilainya lebih kecil dari 0 maka dia akan dijadikan `imePopup.choices.length - 1` dan jika nilainya lebih besar dari `imePopup.choices.length - 1` maka nilainya akan dijadikan 0.

Setelah pembungkusan masukan, tampilan pada kotak pilihan dan kotak keluaran akan diubah sehingga sesuai dengan indeks yang dipilih. Nilai variabel `imePopup.highlighted` juga akan disesuaikan.

3.5.11.8 Fungsi `imePopup.updateCachePriority()`

Fungsi ini akan mengubah urutan transliterasi pada *cache* agar alternatif yang sedang dipilih (`imePopup.converted`) diletakkan menjadi urutan teratas. Fungsi ini dipanggil setiap pengguna selesai memilih suatu transliterasi dari daftar yang ada. Variabel yang terpengaruh adalah `imePopup.cache[imePopup.romaji]`.

3.5.11.9 Fungsi `imePopup.setConverted(str: String)`

Fungsi ini mengubah nilai `imePopup.converted` dan juga tampilannya pada kotak keluaran. Fungsi ini dipanggil setiap pengguna mengetikkan tombol baru di *keyboard* dan juga setiap pengguna memilih alternatif transliterasi.

3.5.11.10 Fungsi imePopup.setRomaji(str: String)

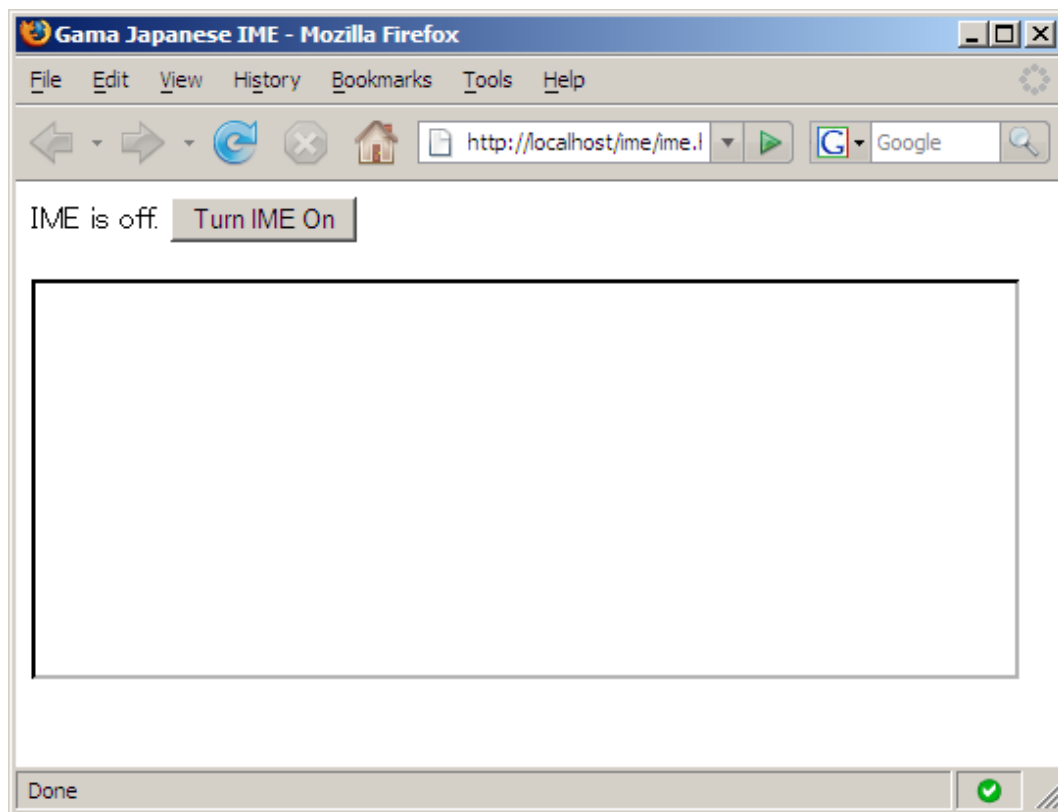
Fungsi ini dipanggil jika masukan *romaji* pengguna berubah. Nilai variabel `imePopup.romaji` akan disesuaikan dan kotak keluaran akan menampilkan *hiragana* dari *romaji* tersebut.

BAB IV

PEMBAHASAN

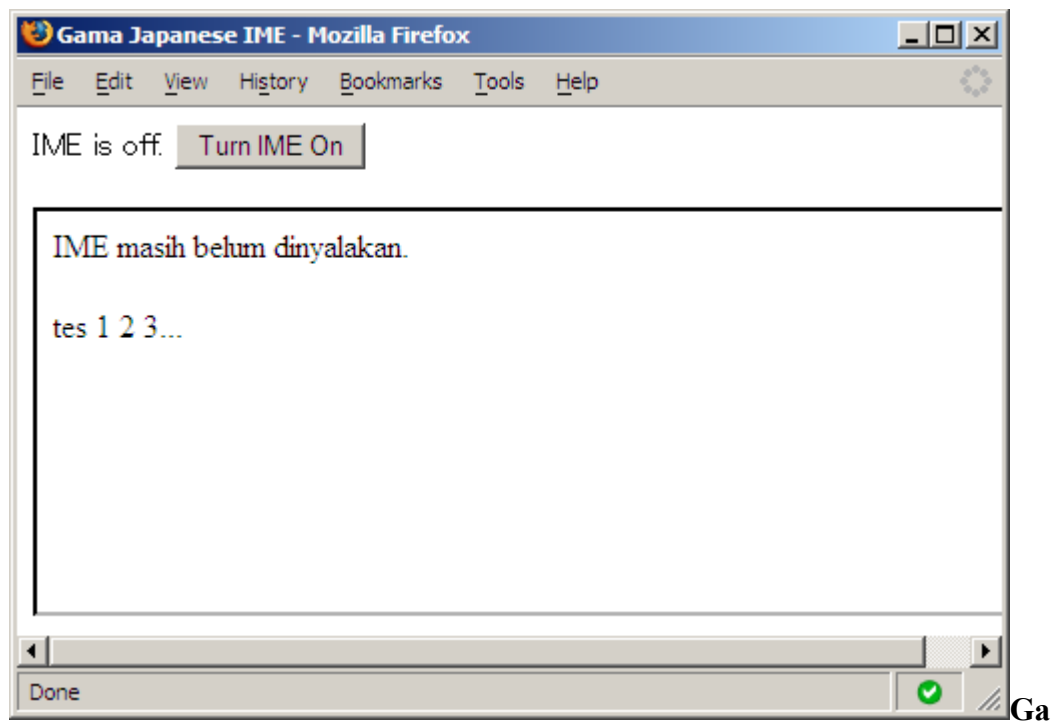
4.1 Antarmuka Aplikasi Web IME

Antarmuka utama aplikasi webnya sangatlah sederhana. Terdapat kotak teks tempat pengguna mengetik, tombol yang menyalakan/mematikan IME, dan status IME:



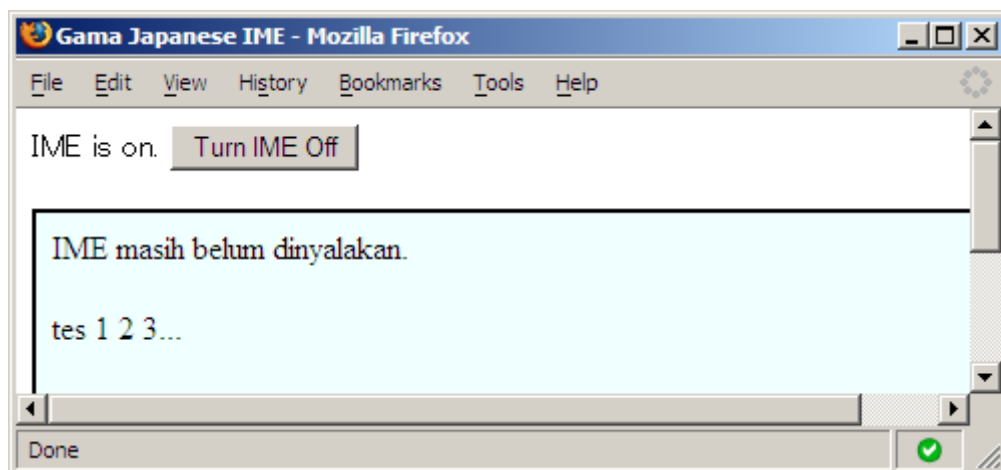
Gambar 4.1 Antarmuka Utama Aplikasi Web IME

Saat halaman pertama dimuat, IME masih belum dinyalakan. Dalam status mati tersebut, kotak teks bertindak layaknya kotak teks pada umumnya. Apa yang diketik pengguna langsung dimasukkan pada kotak teks:



Gambar 4.2 Perilaku Kotak Teks saat IME belum Diaktifkan

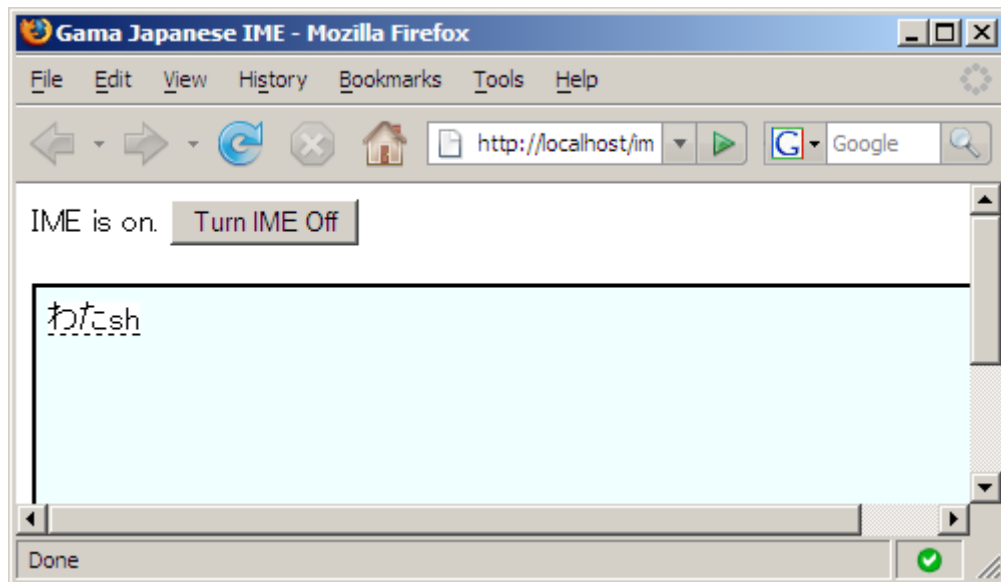
IME bisa diaktifkan dengan mengklik tombol yang ada. Saat IME aktif, latar belakang kotak teks akan diwarnai biru muda untuk menandakan hal tersebut.



Gambar 4.3 IME yang Aktif

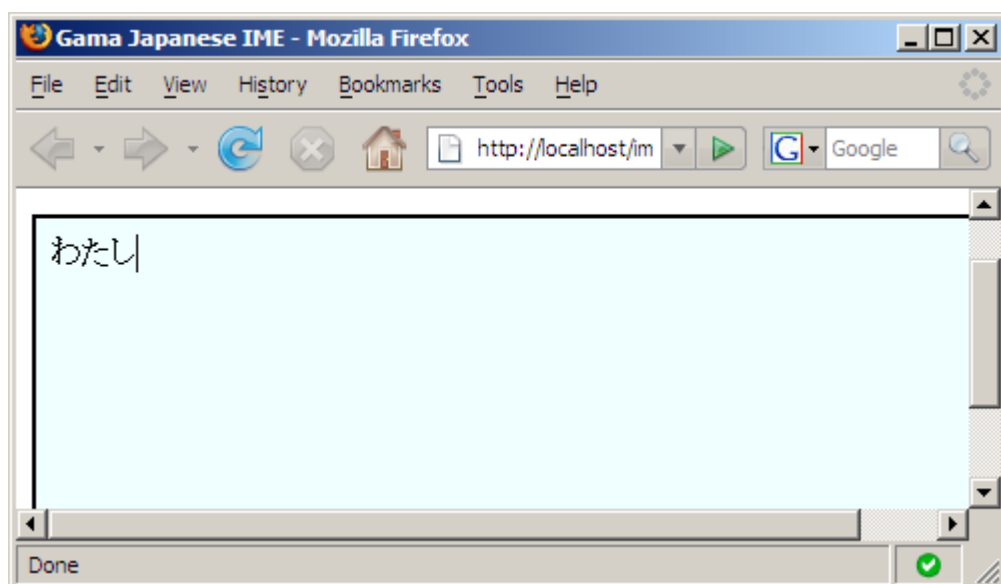
Saat IME aktif, masukan yang dituliskan pengguna tidak langsung

dimasukkan ke dalam kotak teks. Masukan tersebut akan ditampilkan pada *popup* dan ditransliterasi menjadi *hiragana*. Transliterasi tersebut dilakukan di sisi klien:



Gambar 4.4 IME Aktif dengan Masukan 'watash'

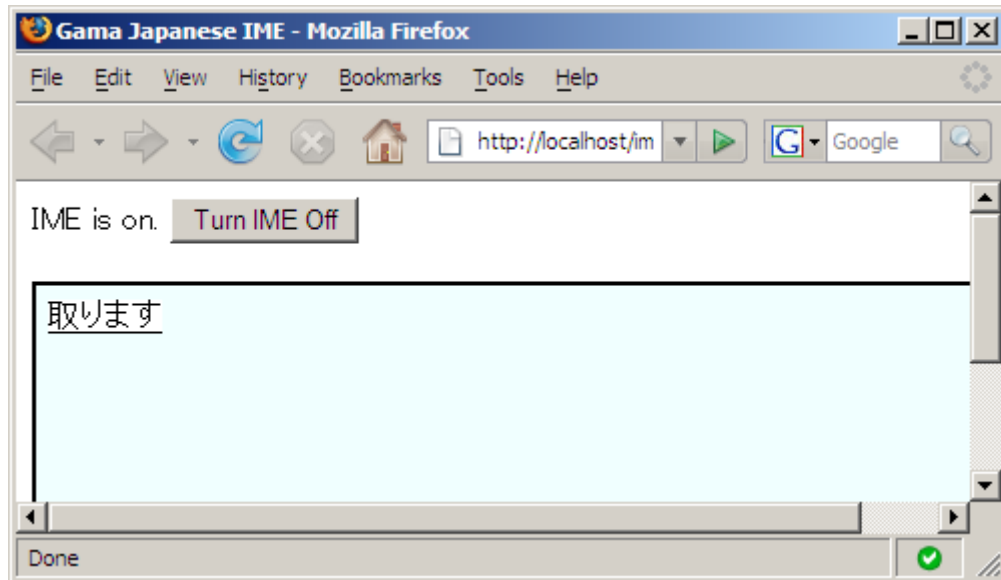
Dengan menekan enter, apa yang tertulis pada popup IME akan dimasukkan ke kotak teks dan input berikutnya siap dimasukkan:



Gambar 4.5 Masukan 'watashi' Dimasukkan ke Kotak Teks

Untuk memperoleh transliterasi *kanji*, setelah masukan dituliskan

pengguna harus menekan spasi. Menggunakan AJAX, transliterasinya akan diminta dari server dan pilihan pertama akan ditampilkan:



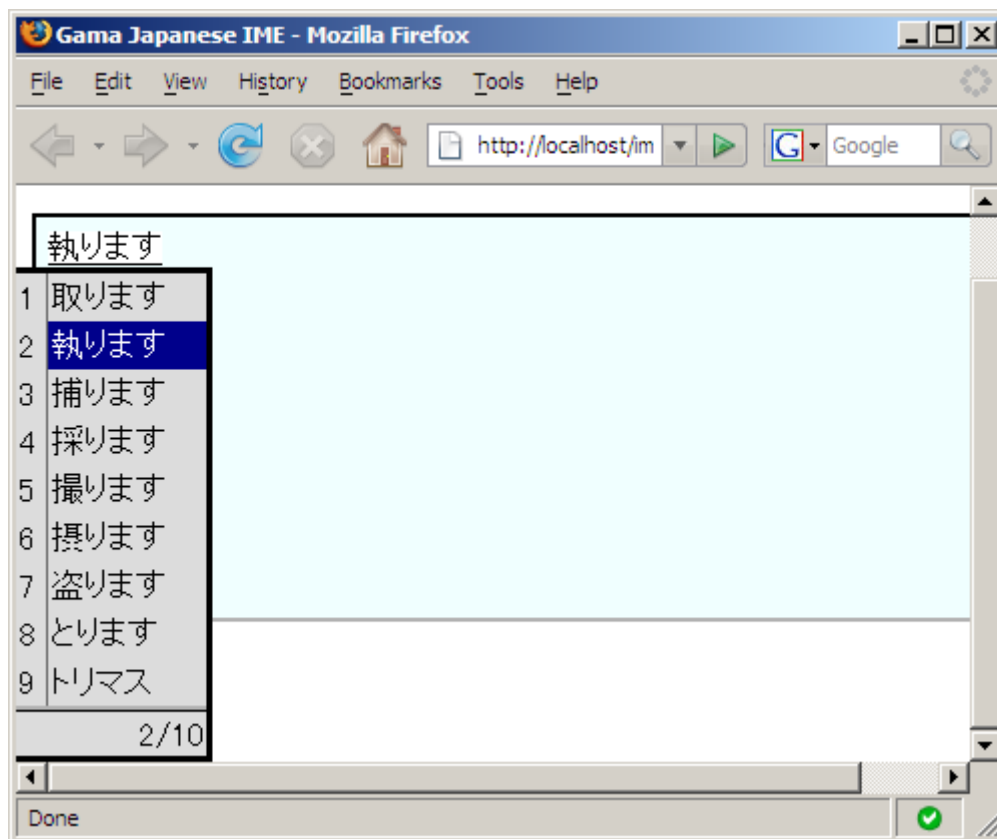
Gambar 4.6 Pilihan Pertama Transliterasi Kanji 'torimasu'

Untuk memilih transliterasi berikutnya, pengguna dapat menekan spasi. Pilihan berikutnya akan dipilih dan *popup* IME akan menampilkan kotak pilihan. Saat kotak pilihan ditampilkan, pengguna dapat menggunakan tombol berikut untuk navigasi pilihan:

Tabel 4.1 Tombol Navigasi Kotak Pilihan

Tombol	Aksi
spasi, bawah	memilih pilihan berikutnya
atas	memilih pilihan sebelumnya
page down	menampilkan halaman pilihan berikutnya
page up	memilih halaman sebelumnya
1-9	memilih pilihan 1 sampai 9

Selain itu, pengguna juga dapat mengklik pilihan yang ada dengan *mouse* untuk memilihnya.

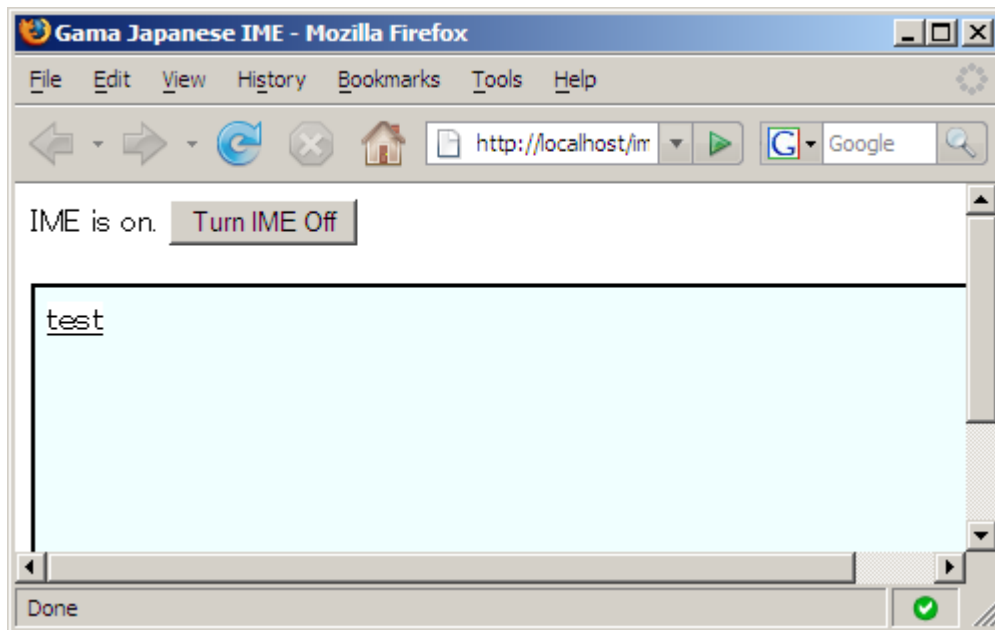


Gambar 4.7 Kotak Pilihan Transliterasi Alternatif untuk 'torimasu'

Pilihan transliterasi *hiragana* dan *katakana* ada pada kotak pilihan. Untuk memperoleh transliterasi khusus seperti *katakana* lebar setengah, diperlukan tombol konversi paksa. Daftar konversi pakasanya adalah sebagai berikut:

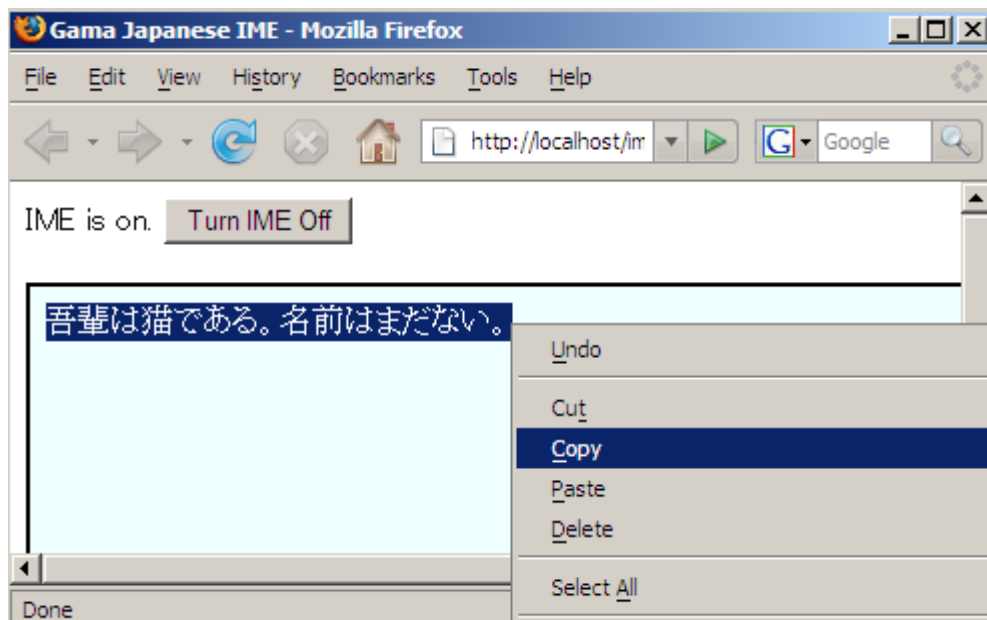
Tabel 4.2 Tombol Konversi Paksa IME

Tombol	Hasil Konversi	Contoh
F6	hiragana	わたし
F7	katakana lebar penuh	ワタシ
F8	katakana lebar setengah	ワタシ
F9	romaji lebar penuh	w a t a s h i
F10	romaji lebar setengah	watashi



Gambar 4.8 Masukan 'test' yang Ditransliterasi Paksa Menjadi Huruf Latin Lebar Penuh

Setelah pengguna selesai menulis, tulisan tersebut bisa dicopy-paste ke tujuan lain.



Gambar 4.9 Menyalin Tulisan untuk Aplikasi Lain

4.2 Keterbatasan Masukan Transliterasi

Secara umum, mesin transliterasi tidak bisa langsung mentransliterasi kalimat kompleks seperti 吾輩は猫である (wagahaihanekodearu). Ini karena mesin transliterasi tidak memiliki komponen untuk memecah kalimat. Dalam contoh kalimat yang diberikan, mesin transliterasi tidak bisa memecah string masukan menjadi, misalnya, 'wagahai', 'ha', 'neko', 'de', dan 'aru'.

Secara teknis, mesin transliterasi hanya bisa menangani modifikasi suatu kata dasar. Modifikasi tersebut adalah penambahan awalan atau infleksi akhir. Jadi, pengguna harus melakukan transliterasi pada unit-unit yang secara teknis bisa ditangani mesin transliterasi. Ilustrasi kasus-kasus yang bisa ditangani mesin transliterasi adalah sebagai berikut:

- Kata dasar, misalnya 猫 'neko', 食べる 'taberu', dan 美しい 'utsukushii'.
- Infleksi dari kata dasar, misalnya 食べない 'tabenai' (bentuk negatif), 食べましょう 'tabemashou' (bentuk hortatif sopan), 食べさせられた 'tabesaserareta' (bentuk kausatif pasif lampau), dan 学生だった 'gakuseidatta' (bentuk lampau).
- Penambahan partikel, misalnya 私は 'watashiha' (partikel topik), 先生の 'senseino' (partikel kepemilikan), dan 本を 'honwo' (partikel objek langsung).
- Penambahan gobi, misalnya 行くぞ 'ikuzo', 素敵ね 'sutekine', dan 痛いよ 'itaiyo'.
- Penambahan prefix, misalnya 大魔王 'daimaou' dan 新メンバー

3	'ashitakouenniikimashouka' 明日公園に行きましようか	'ashita' 明日
		'kouen'ni' 公園に
		'ikimashouka' 行きましようか
4	'yokohamashi(kanagawaken):yaku360man'nin 横浜市（神奈川県）：約360万人	'yoko' 横
		'hama' 浜
		'shi' 市
		(' (
		'kanagawaken' 神奈川県
		'):') :
		'yaku' 約
		'360' 360
		'man' 万
'nin' 人		
5	'kinoutanakasantoaimashitayo' 昨日田中さんと会いましたよ	'kinou' 昨日
		'ta' 田
		'naka' 中
		'santo' さんと
		'aimashitayo' 会いましたよ
6	'watashinonamaehaagurodesu' 私の名前はアグロです	'watashino' 私の
		'namaecha' 名前は
		'aguro' アグロ
		'desu' です

Dari contoh-contoh di atas, bisa ditemukan beberapa keterbatasan sistem yang sebetulnya bisa diatasi tanpa perlu mengembangkan sistem pemecah kalimat:

1. Ketidakmampuan mentransliterasi nama tempat seperti 'yokohamashi' (contoh 4) dan 'tanaka' (contoh 5). Pengguna harus melakukan pemecahan yang merepotkan yaitu 'yoko'+ 'hama'+ 'shi' dan

'ta'+naka'. Transliterasi menjadi per karakter, bukan per kata. Ini karena kamus EDICT secara umum tidak memuat entri nama tempat dan orang. Untuk mengatasi hal ini, perlu diintegrasikan entri dari suatu kamus nama (misal ENAMDICT).

2. Ketidakmampuan menuliskan karakter-karakter tertentu berdasarkan bacaan *on*-nya. Bacaan *on* adalah bacaan suatu kanji yang umum digunakan pada gabungan kanji. Sebagai contoh (contoh 2), pada 'nihonkoku' 日本国 kanji 国 dibaca 'koku'. Pengguna tidak bisa memberikan masukan 'nihonkoku' langsung, karena entri tersebut tidak ada di EDICT. Namun pengguna juga tidak bisa memecahnya menjadi 'nihon'+koku', sebab pada EDICT 国 yang berdiri sendiri dibaca sebagai 'kuni', bacaan *kun*-nya. Jadi untuk menuliskan 'nihonkoku' pengguna harus menuliskan 'nihon'+kuni'. Untuk mengatasi hal ini, perlu diintegrasikan informasi bacaan *on* semua kanji dari suatu kamus kanji (misal KANJIDIC).
3. Ketidakmampuan menyimpulkan keberadaan kata yang ditulis dengan katakana. Pada contoh 6, 'agurodesu' harus dipisah menjadi 'aguro'+desu'. Ini karena mesin transliterasi sebetulnya bisa menyimpulkan bahwa kata dasar dari 'agurodesu' adalah 'aguro', namun pada EDICT tidak terdapat kata 'aguro'. Untuk mengatasi hal ini, mesin transliterasi harus melacak apakah suatu kata dasar hanyalah diberi awalan dan akhiran (tidak diinfleksi), dan jika ya maka memberikan transliterasi *katakana* otomatis atas kata dasar tersebut.

4.3 Kekurangan Algoritma Transliterasi

4.3.1 Kemampuan Mengurutkan Pilihan yang Terbatas

Algoritma transliterasi hanya memanfaatkan data kepopuleran pada EDICT untuk mengurutkan hasil transliterasinya. Padahal, data kepopuleran yang diberikan EDICT sangatlah kasar. Secara khusus, hanya ada satu penanda yaitu P yang menunjukkan bahwa suatu entri EDICT populer. Algoritma transliterasi berusaha untuk melakukan pemisahan lebih lanjut dengan memanfaatkan penanda-penanda seperti $\circ K$ (3.2.4.6).

Dari penanda-penanda yang ada, diperoleh jangkauan kepopuleran -2 sampai 2. Bisa dilihat bahwa algoritma transliterasi tidak bisa melakukan apa-apa jika terdapat banyak pilihan dengan kepopuleran yang sama. Sebagai contoh, untuk transliterasi 'tatsu' 3 pilihan pertamanya berturut-turut adalah 建つ, 断つ, dan 立つ. Ketiga kata tersebut pada EDICT memiliki penanda P sehingga semuanya mendapat nilai kepopuleran 2, dan urutan di atas hanyalah kebetulan yang bergantung pada urutan entri-entrinya di berkas EDICT. Padahal, idealnya 立つ (berdiri) diberikan sebagai pilihan pertama karena paling umum.

Solusinya adalah mengintegrasikan data kepopuleran dari kamus *kanji* yang lebih granular. Sebagai contoh, menggunakan kamus kanji KANJIDIC bisa didapatkan informasi bahwa kanji 立 berada di urutan 58 sedangkan 建 berada di urutan 300 dan 断 pada urutan 338.

4.3.2 Kekurangan Mesin Transliterasi yang Berbasis Romaji

Aturan tata bahasa dispesifikasikan menggunakan *romaji* karena banyak aturan infleksi yang bisa dinyatakan secara lebih sederhana menggunakan *romaji*. Untuk mempermudah implementasi, mesin transliterasinya secara internal juga menggunakan *romaji*.

Namun ini menyebabkan diperlukannya transliterasi dari *romaji* ke *kana* untuk setiap perantaraan, karena pada akhirnya diperlukan *kana*-nya untuk query ke basis data. Proses ini dapat dihilangkan jika mesin transliterasi secara internal sepenuhnya berbasis *kana*. Dengan begitu performa transliterasi akan meningkat karena satu tahap bisa dihilangkan.

Jika mesin transliterasi sepenuhnya berbasis *kana*, namun diinginkan penulisan aturan yang berbasis *romaji*, maka tantangannya adalah mengkonversi aturan berbasis *romaji* tersebut menjadi aturan internal berbasis *kana*. Prosesnya tidak sesederhana konversi string biasa karena diperlukan pemahaman mengenai fonologi bahasa Jepang.

Sebagai contoh, aturan berikut:

```
v5s, v5k, v5k-s, v5g, v5b, v5t, v5m, v5r, v5aru, v5n, v5u, v5u-
s/*u/*i/stem, n
```

Harus dikonversi menjadi:

```
v5s/*す/*し/stem, n
v5k/*く/*き/stem, n
...
```

Bisa dilihat bahwa walaupun aturan hanya menspesifikasikan bahwa “ubah akhiran ‘i’ menjadi ‘u’”, dalam proses konversi program harus cerdas dan

tahu bahwa untuk $v5s$, yang dimaksud adalah merubah す 'su' menjadi し 'si', dst.

Mesin transliterasi berbasis *kana* juga tidak akan menemukan kata dasar gadungan tertentu yang tidak bisa terdeteksi oleh pengecekan fonologis yang saat ini digunakan. Contohnya, dalam perantaraan 'taberu' akan ditemukan kata dasar $v5u$ 'tabu'. Jika dilihat dari sudut pandang *romaji*, memang benar bahwa 'tabu' berakhiran 'u', namun jika dilihat dari sudut pandang *kana* terlihat jelas bahwa たぶ (tabu) tidak mungkin merupakan $v5u$ karena berakhiran ぶ (bu) bukannya う (u). Jika menggunakan mesin berbasis *kana* maka $v5u$ たぶ tidak akan mungkin terantai dari awal.

BAB V

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Dari laporan penulisan tugas akhir ini dapat ditarik beberapa kesimpulan tentang sistem yang dibangun, yaitu:

1. Sistem terdiri dari 4 bagian, yaitu aplikasi web IME klien, aplikasi perantara ASP.NET `transliterator.aspx`, mesin pentransliterasi `GamaIme.dll`, dan program pembangun basis data `EdictDbBuilder.exe`.
2. Sistem dapat melakukan transliterasi kata dasar bahasa Jepang, kata yang terinfleksi, kata yang diberi awalan, partikel, maupun gobi, dan pola-pola kalimat umum tertentu. Kombinasi dari bentuk-bentuk tersebut juga bisa ditangani.
3. Hasil transliterasi diurutkan berdasarkan data kepopuleran yang terbatas yang ada pada EDICT.
4. Permintaan transliterasi dilakukan menggunakan AJAX dan hasilnya disimpan di *cache* aplikasi web.

5.2 Saran

Inilah saran untuk meningkatkan fungsionalitas maupun performa sistem:

1. Menambahkan lapisan pemecah kalimat di sisi server, sehingga kalimat sepanjang dan sekompleks apapun dapat ditangani. Sisi klien juga memerlukan modifikasi sehingga jika pemecahan yang dilakukan sistem

tidak sesuai keinginan pemakai, pemakai dapat memodifikasi batas katanya. Lapisan pemecah kalimat ini akan membuat fungsionalitas sistemnya setara dengan IME-IME kelas tinggi desktop.

2. Mengubah sistem transliterasinya agar secara internal sepenuhnya menggunakan *kana*. Ini dapat meningkatkan performa sebagaimana dijelaskan di 4.3.2. Untuk memungkinkan hal ini, sebelumnya perlu dikembangkan sistem yang bisa menerjemahkan aturan berbasis *romaji* menjadi aturan berbasis *kana*.
3. Mengintegrasikan data kepopuleran *kanji* pada kamus seperti KANJIDIC, sebagaimana dijelaskan pada 4.3.1. Dengan ini urutan pilihan transliterasi akan lebih baik.
4. Mengintegrasikan data kamus nama orang dan tempat seperti ENAMDICT untuk mengatasi kekurangan yang dideskripsikan pada 4.2 contoh 1.
5. Mengintegrasikan data bacaan *on kanji* seperti pada KANJIDIC untuk mengatasi kekurangan yang dideskripsikan pada 4.2 contoh 2.
6. Menambahkan kemampuan mengenali kata dasar katakana yang tidak terdapat di kamus sebagaimana dideskripsikan pada 4.2 contoh 3.

DAFTAR PUSTAKA

- , *CSS Tutorial*, <http://w3schools.com/css/default.asp>, diakses April 2007
- , Google Suggest, <http://suggest.google.com/>, diakses April 2007
- , *HTML Tutorial*, <http://w3schools.com/html/default.asp>, diakses April 2007
- , *Japanese writing system*,
http://en.wikipedia.org/wiki/Japanese_writing_system, diakses April 2007
- , Meebo, <http://www.meebo.com/>, diakses April 2007
- , *SCIM*, <http://www.scim-im.org/>, diakses April 2007
- , *SQLite home page*, <http://sqlite.org/>, diakses April 2007
- , *System.Data.SQLite*, <http://sqlite.phxsoftware.com/>, diakses April 2007
- , *What is an IME (Input Method Editor) and how do I use it?*,
http://www.microsoft.com/globaldev/handson/user/IME_Paper.msp, diakses April 2007
- , *Why use AJAX?*,
www.interaktonline.com/Support/Articles/Details/AJAX:+Asynchronously+Moving+Forward-Why+use+AJAX%3F.html?id_art=36&id_asc=309
- Breen, J., *The EDICT Dictionary File*,
http://www.csse.monash.edu.au/~jwb/j_edict.html, diakses April 2007

- Cho, I., 2006, *Web-tier Programming Codecamp II*, Sun Microsystems.
Inc
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., dan Stein, C., 2001,
Introduction to Algorithms, The MIT Press
- ECMA-334 committee, *ECMA-334 C# Language Specification*,
<http://www.ecma-international.org/publications/files/ecma-st/ECMA-334.pdf>
- Elias, A., *A complete introduction to Japanese character encodings*,
<http://www.cs.mcgill.ca/~aelias4/>, diakses April 2007
- Garrett, J. J., 2005, *Ajax : A New Approach to Web Applications*,
<http://www.adaptivepath.com/publications/essays/archives/000385.php>, diakses April 2007
- Kaiser, S., Ichikawa, Y., Kobayashi, N., dan Yamamoto, H., 2001,
Japanese, A Comprehensive Grammar, Routledge, London
- Kim, T., *A Japanese Guide to Japanese Grammar*,
<http://www.guidetojapanese.org>, diakses April 2007
- Kudo, T., *Ajax IME: Web-based Japanese Input Method*,
<http://ajaxime.chasen.org/>, diakses April 2007
- Resig, J., 2006, *Pro JavaScript Techniques*, APress, USA
- Schildt, H., 2001, *C#: A Beginner's Guide*, McGraw-Hill Book Company,
New York
- Seeley, C., 1991, *A History of Writing in Japan*, University of Hawai'i
Press
- Tien, T. S., 2001, *Bahasa C# untuk Pemrograman Berorientasi Objek*, PT
Elex Media Komputindo, Jakarta
- Walther, S., 2006, *ASP.NET 2.0 Unleashed*, Sams

Yoshida, Y., 1996, *Japanese for Today (Bahasa Jepang Sehari-hari)*, PT

Gramedia Widiasarana Indonesia, Jakarta

Zakas, N. C., 2005, *Professional JavaScript for Web Developers*, Wiley

Publishing, Inc., Indianapolis

Daftar Isi

BAB I PENDAHULUAN.....	1
1.1 Latar Belakang Masalah.....	1
1.2 Rumusan Masalah.....	3
1.3 Batasan Masalah.....	3
1.4 Tujuan Penelitian.....	4
1.5 Manfaat Penelitian.....	5
1.6 Tinjauan Pustaka.....	5
1.7 Metodologi Penelitian.....	6
1.8 Sistematika Penulisan.....	7
BAB II LANDASAN TEORI.....	9
2.1 Sistem Tulis Bahasa Jepang.....	9
2.1.1 Hiragana.....	9
2.1.1.1 Simbol-Simbol Hiragana.....	10
2.1.1.2 Penggunaan Hiragana.....	15
2.1.2 Katakana.....	16
2.1.2.1 Simbol-Simbol Katakana.....	16
2.1.2.2 Penggunaan Katakana.....	19
2.1.3 Kanji.....	21
2.1.3.1 Bacaan on.....	22
2.1.3.2 Bacaan kun.....	23
2.1.3.3 Bacaan nanori.....	24
2.1.3.4 Bacaan yang digunakan.....	24
2.2 Transliterasi Bahasa Jepang menggunakan Huruf Latin.....	26
2.3 Modifikasi Kata Dasar pada Bahasa Jepang.....	30
2.4 Partikel dan Gobi.....	33
2.5 Input Method Editor (IME).....	33
2.5.1 IME Fonetis Windows XP.....	36
2.6 Himpunan Karakter dan Pengkodean Bahasa Jepang di Komputer.....	38
2.6.1 Himpunan Karakter dan Pengkodean.....	38
2.6.2 Himpunan Karakter Bahasa Jepang.....	39
2.6.2.1 Hiragana pada Unicode.....	40
2.6.2.2 Katakana pada Unicode.....	40
2.6.2.3 Kanji pada Unicode.....	42
2.7 Breadth-first Search (BFS).....	43
2.8 EDICT.....	43
2.8.1 Format.....	44
2.9 SQLite.....	48
2.10 Framework .NET, C#, dan ASP.NET.....	49
2.10.1 Framework .NET.....	49
2.10.2 C#.....	50
2.10.3 ASP.NET.....	51
2.11 JavaScript dan AJAX.....	52
2.11.1 JavaScript.....	52

2.11.2 AJAX.....	52
2.12 Pustaka Event Dean Edwards.....	55
BAB III ANALISIS, PERANCANGAN, DAN IMPLEMENTASI.....	56
3.1 Gambaran Umum Sistem.....	56
3.2 Komponen Pembangun Basis Data: EdictDbBuilder.exe.....	58
3.2.1 Tinjauan.....	58
3.2.2 Analisis Permasalahan.....	58
3.2.3 Analisis Kebutuhan Sistem.....	59
3.2.4 Tabel edict.....	59
3.2.4.1 Field kana.....	60
3.2.4.2 Field kanji.....	60
3.2.4.3 Contoh Pengisian (kana, kanji).....	60
3.2.4.4 (kana, kanji) sebagai Kunci Primer.....	62
3.2.4.5 Field tags, edict_allowed_tags.txt, dan edict_tags_transform.txt.....	63
3.2.4.6 Field pop.....	67
3.2.4.7 Field uk.....	70
3.2.5 Implementasi.....	70
3.2.6 Kelas EdictEntry.....	71
3.2.6.1 Tinjauan.....	71
3.2.6.2 Perancangan Kelas.....	71
3.2.6.3 Metode Statik string EdictEntry.KatakanaToHiragana(string input).....	72
3.2.6.4 Metode void EdictEntry.TryAddTag(List<string> tagList, string tag).....	74
3.2.6.5 Konstruktors EdictEntry().....	74
3.2.7 Kelas BuildDb.....	77
3.2.7.1 Tinjauan.....	77
3.2.7.2 Perancangan Kelas.....	77
3.2.7.3 Metode Statik string BuildDb.JoinTags(string[] tags).....	78
3.2.7.4 Metode Statik void BuildDb.Help().....	78
3.2.7.5 Metode Statik void BuildDb.PopulateDb(SQLiteConnection conn, List<EdictEntry> entries).....	78
3.2.7.6 Metode Statik int BuildDb.Main(string[] args).....	79
3.3 Komponen Pentransliterasi Kanji: GamaIme.dll.....	80
3.3.1 Tinjauan.....	80
3.3.2 Analisis Permasalahan.....	80
3.3.3 Analisis Kebutuhan Sistem.....	80
3.3.4 Mesin Transliterasi Kanji.....	81
3.3.4.1 Tinjauan.....	81
3.3.4.2 Penggunaan Romaji.....	84
3.3.4.3 Romanisasi yang Digunakan.....	85
3.3.4.4 Representasi Aturan.....	88
3.3.4.5 Aplikasi Aturan pada Kata Nyata.....	92
3.3.4.6 Algoritma Perantaraan.....	94
3.3.4.7 Algoritma Breadth-first Search.....	106
3.3.5 Mesin Transliterasi Romaji ke Kana dan Sebaliknya.....	107

3.3.5.1 Tinjauan.....	107
3.3.5.2 Algoritma.....	108
3.3.5.3 Implementasi.....	110
3.3.6 Kelas Transliterat<S, D>.....	111
3.3.6.1 Tinjauan.....	111
3.3.6.2 Perancangan Kelas.....	111
3.3.6.3 Penggunaan.....	112
3.3.7 Kelas StringTransliterat.....	116
3.3.7.1 Tinjauan.....	116
3.3.7.2 Perancangan Kelas.....	116
3.3.7.3 Metode Statik ParseRule(string rules).....	117
3.3.8 Kelas RomajitoKanaTransliterat.....	117
3.3.8.1 Tinjauan.....	117
3.3.8.2 Perancangan Kelas.....	118
3.3.8.3 Contoh Penggunaan.....	118
3.3.9 Kelas HiraganaToRomajiTransliterat.....	118
3.3.9.1 Tinjauan.....	119
3.3.9.2 Perancangan Kelas.....	119
3.3.9.3 Contoh Penggunaan.....	119
3.3.10 Kelas RuleInstance.....	120
3.3.10.1 Tinjauan.....	120
3.3.10.2 Perancangan Kelas.....	121
3.3.10.3 Ilustrasi Nilai Atribut.....	122
3.3.10.4 Metode string RuleInstance.ToString().....	125
3.3.10.5 Kontruktor RuleInstance().....	126
3.3.10.6 Kontruktor RuleInstance(string perfectMatch).....	126
3.3.10.7 Metode List<string> Replace(string dictionaryWord).....	126
3.3.11 Kelas Rule.....	130
3.3.11.1 Tinjauan.....	130
3.3.11.2 Perancangan Kelas.....	131
3.3.11.3 Metode Rule[] Load(string ruleFile).....	133
3.3.12 Kelas RestrictionChecker.....	133
3.3.12.1 Tinjauan.....	133
3.3.12.2 restriction.txt.....	135
3.3.12.3 Perancangan Kelas.....	137
3.3.12.4 Metode Statik void Load(string fileName).....	137
3.3.12.5 Contoh Penggunaan.....	139
3.3.13 Kelas KanjiTransliterat.....	139
3.3.13.1 Tinjauan.....	139
3.3.13.2 Perancangan Kelas.....	139
3.3.13.3 Contoh Penggunaan.....	141
3.3.13.4 Kontruktor KanjiTransliterat(string ruleFileName, string restrictionFileName, string tagFileName, string dbFileName).....	141
3.3.13.5 Metode string[] KanjiTransliterat.Transliterate(string inputRomaji).....	142
3.3.13.6 Metode string[] FetchDb(List<List<RuleInstance>>	

instancesByLevel).....	142
3.4 Komponen Perantara Klien Server: transliterator.aspx.....	143
3.5 Komponen Aplikasi Web IME.....	144
3.5.1 Tinjauan.....	144
3.5.2 Analisis Permasalahan.....	144
3.5.3 Analisis Kebutuhan Sistem.....	145
3.5.4 Perancangan Antarmuka.....	145
3.5.5 Penanganan Keyboard.....	148
3.5.6 Meminta Transliterasi ke Server dengan AJAX.....	150
3.5.7 Implementasi.....	150
3.5.8 misc.js.....	150
3.5.9 transliterator.js.....	152
3.5.10 keyboard.js.....	153
3.5.11 ime.js.....	154
3.5.11.1 Diagram Kelas dan Atribut ime.....	154
3.5.11.2 Diagram Kelas dan Atribut imePopup.....	156
3.5.11.3 Inisialisasi ime.....	158
3.5.11.4 Penanganan Keyboard.....	159
3.5.11.5 Permintaan Transliterasi ke Server dan Caching.....	160
3.5.11.6 Fungsi imePopup.init.....	161
3.5.11.7 Fungsi imePopup.setHighlighted(index: Number).....	161
3.5.11.8 Fungsi imePopup.updateCachePriority().....	162
3.5.11.9 Fungsi imePopup.setConverted(str: String).....	162
3.5.11.10 Fungsi imePopup.setRomaji(str: String).....	163
BAB IV PEMBAHASAN.....	164
4.1 Antarmuka Aplikasi Web IME.....	164
4.2 Keterbatasan Masukan Transliterasi.....	170
4.3 Kekurangan Algoritma Transliterasi.....	174
4.3.1 Kemampuan Mengurutkan Pilihan yang Terbatas.....	174
4.3.2 Kekurangan Mesin Transliterasi yang Berbasis Romaji.....	174
BAB V KESIMPULAN DAN SARAN.....	177
5.1 Kesimpulan.....	177
5.2 Saran.....	177
DAFTAR PUSTAKA.....	179

Daftar Gambar

Gambar 2.1 Kedudukan IME pada proses masukan.....	35
Gambar 2.2 Konversi “mita” pada IME.....	37
Gambar 2.3 Hasil konversi “mita”.....	37
Gambar 2.4 Tampilan kemungkinan “mita”.....	38
Gambar 2.5 Hasil konversi “mita” yang telah dikirim ke aplikasi.....	38
Gambar 2.6 Hello World di C#.....	51
Gambar 2.7 Contoh aplikasi ASP.NET.....	51
Gambar 2.8 Model aplikasi web klasik.....	53
Gambar 2.9 Model aplikasi web menggunakan AJAX.....	54
Gambar 3.1 Hubungan Komponen IME.....	57
Gambar 3.2 edict_allowed_tags.txt.....	65
Gambar 3.3 edict_tags_transform.txt.....	67
Gambar 3.4 edict_popularity_tags.txt.....	69
Gambar 3.5 Diagram Kelas EdictEntry.....	72
Gambar 3.6 Metode KatakanaToHiragana.....	73
Gambar 3.7 Pemisahan Arti pada Konstruktor EdictEntry.....	75
Gambar 3.8 Pengisian Bacaan dan Penulisan pada Konstruktor EdictEntry.....	76
Gambar 3.9 Pengisian Penanda pada Konstruktor EdictEntry.....	77
Gambar 3.10 Diagram Kelas BuildDb.....	78
Gambar 3.11 Penggunaan BEGIN dan QUERY pada PopulateDb.....	79
Gambar 3.12 Contoh Informasi pada suatu Rule.....	82
Gambar 3.13 Contoh Informasi pada suatu RuleInstance.....	82
Gambar 3.14 Contoh Informasi pada suatu RuleInstance yang diperoleh melalui Perantaraan.....	82
Gambar 3.15 Contoh Informasi pada suatu RuleInstance untuk Dirantarkan.....	83
Gambar 3.16 Contoh Informasi pada suatu RuleInstance hasil Perantaraan.....	83
Gambar 3.17 Pohon Perantaraan “tabenakat'ta”.....	84
Gambar 3.18 Algoritma Perantaraan bagian 1.....	95
Gambar 3.19 Ilustrasi Penanda yang tidak Cocok dalam Perantaraan.....	96
Gambar 3.20 Algoritma Perantaraan bagian 2.....	96
Gambar 3.21 Ilustrasi Kasus Perantaraan Kata Kosong yang Ingin Dicegah.....	97
Gambar 3.22 Algoritma Perantaraan bagian 3.....	97
Gambar 3.23 Ilustrasi Kasus Perantaraan yang Depannya Cocok.....	97
Gambar 3.24 Algoritma Perantaraan bagian 4.....	97
Gambar 3.25 Ilustrasi Kasus Perantaraan yang Belakangnya Cocok.....	98
Gambar 3.26 Algoritma Perantaraan bagian 5.....	98
Gambar 3.27 Pemberian Nilai srcTag pada nextInstance.....	98
Gambar 3.28 Algoritma Perantaraan bagian 6.....	98
Gambar 3.29 Penentuan Kata Dasar nextInstance.....	99
Gambar 3.30 Algoritma Perantaraan bagian 7.....	99
Gambar 3.31 Contoh Kasus yang Melanggar Syarat Fonologis Penanda.....	99
Gambar 3.32 Algoritma Perantaraan bagian 8.....	99
Gambar 3.33 Algoritma Perantaraan bagian 9.....	100

Gambar 3.34 Contoh Perantaian Bagian Depan Kana.....	100
Gambar 3.35 Algoritma Perantaian bagian 10.....	100
Gambar 3.36 Contoh Perantaian Bagian Depan Kanji.....	101
Gambar 3.37 Algoritma Perantaian bagian 11.....	102
Gambar 3.38 Algoritma Perantaian bagian 12.....	102
Gambar 3.39 Algoritma Perantaian bagian 13.....	103
Gambar 3.40 Contoh Penentuan Nilai endOrigKana pada Perantaian.....	103
Gambar 3.41 Algoritma Perantaian bagian 14.....	103
Gambar 3.42 Contoh Penentuan Nilai endNewKana pada Perantaian.....	104
Gambar 3.43 Algoritma Perantaian bagian 15.....	104
Gambar 3.44 Contoh Penentuan Nilai endNewKanji pada Perantaian karena rule	105
Gambar 3.45 Algoritma Perantaian bagian 16.....	105
Gambar 3.46 Contoh Penentuan Nilai endNewKanji pada Perantaian karena instance.....	105
Gambar 3.47 Contoh Pohon Aturan Transliterasi Kana.....	110
Gambar 3.48 Diagram Hubungan berbagai Kelas Pentransliterasi Kana.....	111
Gambar 3.49 Diagram Kelas Transliterator.....	112
Gambar 3.50 Diagram Kelas StringTransliterator.....	116
Gambar 3.51 Diagram Kelas RomajiToKanaTransliterator.....	118
Gambar 3.52 Contoh Penggunaan RomajiToKanaTransliterator.....	118
Gambar 3.53 Diagram Kelas HiraganaToRomajiTransliterator.....	119
Gambar 3.54 Contoh Penggunaan HiraganaToRomajiTransliterator.....	119
Gambar 3.55 Diagram Kelas RuleInstance.....	121
Gambar 3.56 Ilustrasi middle untuk soundMorph false.....	128
Gambar 3.57 Ilustrasi middle untuk soundMorph true.....	129
Gambar 3.58 Ilustrasi hasil infleksi untuk soundMorph false.....	130
Gambar 3.59 Ilustrasi hasil infleksi untuk soundMorph true.....	130
Gambar 3.60 Diagram Kelas Rule.....	131
Gambar 3.61 restriction.txt.....	136
Gambar 3.62 Diagram Kelas RestrictionChecker.....	137
Gambar 3.63 Contoh Penggunaan RestrictionChecker.....	139
Gambar 3.64 Diagram Kelas KanjiTransliterator.....	140
Gambar 3.65 Contoh Penggunaan Kelas KanjiTransliterator.....	141
Gambar 3.66 Perancangan Antarmuka Utama Aplikasi Web IME.....	146
Gambar 3.67 Perancangan Antarmuka Popup IME.....	146
Gambar 3.68 Fungsi next yang mengabaikan elemen teks.....	151
Gambar 3.69 Diagram Kelas ime.....	155
Gambar 3.70 Diagram Kelas imePopup.....	156
Gambar 3.71 Kode writeIme yang Menuliskan IME-nya.....	158
Gambar 3.72 Kode HTML Popup IME.....	159
Gambar 3.73 Pengecekan Cache setelah Respons AJAX diterima.....	161
Gambar 3.74 Perpindahan ke Mode selecting setelah Respons AJAX diterima.	161
Gambar 4.1 Antarmuka Utama Aplikasi Web IME.....	164
Gambar 4.2 Perilaku Kotak Teks saat IME belum Diaktifkan.....	165
Gambar 4.3 IME yang Aktif.....	165

Gambar 4.4 IME Aktif dengan Masukan 'watah'.....	166
Gambar 4.5 Masukan 'watashi' Dimasukkan ke Kotak Teks.....	166
Gambar 4.6 Pilihan Pertama Transliterasi Kanji 'torimasu'	167
Gambar 4.7 Kotak Pilihan Transliterasi Alternatif untuk 'torimasu'.....	168
Gambar 4.8 Masukan 'test' yang Ditransliterasi Paksa Menjadi Huruf Latin Lebar Penuh.....	169
Gambar 4.9 Menyalin Tulisan untuk Aplikasi Lain.....	169

Daftar Tabel

Tabel 2.1 Hiragana Dasar.....	10
Tabel 2.2 Hiragana dengan Dakuten.....	11
Tabel 2.3 Hiragana dengan Handakuten.....	12
Tabel 2.4 Hiragana dengan Youon.....	12
Tabel 2.5 Hiragana untuk Suara Luar.....	14
Tabel 2.6 Katakana Dasar.....	17
Tabel 2.7 Katakana dengan Dakuten.....	17
Tabel 2.8 Katakana dengan Handakuten.....	17
Tabel 2.9 Katakana dengan Youon.....	18
Tabel 2.10 Katakana untuk Suara Luar.....	18
Tabel 2.11 Contoh Bacaan On Beberapa Kanji.....	23
Tabel 2.12 Contoh Bacaan Kun Beberapa Kanji.....	24
Tabel 2.13 Perbedaan konsistensi fonologis Hepburn, Nihon-shiki, dan Kunrei-shiki.....	27
Tabel 2.14 Padanan bahasa Indonesia beberapa infleksi pada bahasa Jepang.....	30
Tabel 2.15 Ilustrasi penulisan “tobu” dan berbagai infleksinya.....	30
Tabel 2.16 Beberapa infleksi “kuru”.....	31
Tabel 2.17 Infleksi lampau berbagai jenis verba.....	32
Tabel 2.18 Hiragana pada Unicode.....	40
Tabel 2.19 Katakana Lebar Penuh pada Unicode.....	42
Tabel 2.20 Katakana Lebar Setengah pada Unicode.....	42
Tabel 2.21 Penanda Kelas Kata EDICT.....	45
Tabel 3.1 Field-Field Tabel edict.....	59
Tabel 3.2 Contoh Masukan dan Keluaran Transliterasi Karakter.....	113
Tabel 3.3 Tabel Perubahan Keadaan Popup IME.....	147
Tabel 3.4 Tombol Konversi Paksa IME.....	148
Tabel 3.5 Objek Tabel Kode Karakter Keyboard pada keyboard.js.....	153
Tabel 4.1 Tombol Navigasi Kotak Pilihan.....	167
Tabel 4.2 Tombol Konversi Paksa IME.....	168
Tabel 4.3 Contoh Pemecahan Kalimat.....	171