MODUL PRAKTIKUM

rpl : teknik berorientasi objek
CSG2I1

IF LAB

Gedung F Lt. 3 IFLAB1-IFLAB4
School of Computing
Telkom University

Telkom University

informatics lab

LEMBAR PENGESAHAN

Saya yang bertanda tangan di bawah ini:
Nama : Dawam Dwi Jatmiko Suwawi, S.T., M.T.
NIP : 10890718-3
Dosen PJMP : RPL : Teknik Berorientasi Objek
KK : SIDE
Menerangkan dengan sesungguhnya bahwa modul ini digunakan untuk pelaksanaan praktikum di semester genap tahun ajaran 2015/2016 di Informatics Lab Fakultas Informatika Universitas Telkom.

Bandung, Januari 2016

Mengesahkan,	Mengetahui,
Dosen PJMP RPL : Teknik Berorientasi Objek	Kaprodi S-1 Fakultas Informatika
Dawam Dwi Jatmiko S.T., M.T.	Moch. Arif Bijaksana, Ph.D

PERATURAN PRAKTIKUM
LABORATORIUM INFORMATIKA 2015/2016

1. Praktikum diampu oleh dosen kelas dan dibantu oleh asisten laboratorium dan asisten praktikum.
2. Praktikum dilaksanakan di Gedung Kultubai Selatan (IFLAB 1 s/d IFLAB 5) sesuai jadwal yang ditentukan.
3. Praktikan wajib membawa modul praktikum, kartu praktikum, dan alat tulis.
4. Praktikan wajib mengisi daftar hadir *rooster* dan BAP praktikum dengan bolpoin bertinta hitam.
5. Durasi kegiatan praktikum S-1 = 3 jam (150 menit).
   a. 15 menit untuk pengerjaan Tes Awal atau wawancara Tugas Pendahuluan
   b. 45 menit untuk penyampaian materi
   c. 90 menit untuk pengerjaan jurnal dan tes akhir
6. Jumlah pertemuan praktikum:
   - 10 kali di lab (praktikum rutin)
   - 3 kali di luar lab (terkait Tugas Besar dan UAS)
   - 1 kali berupa presentasi Tugas Besar atau pelaksanaan UAS
7. Praktikan wajib hadir minimal 75% dari seluruh pertemuan praktikum di lab. Jika total kehadiran kurang dari 75% maka nilai UAS/ Tugas Besar = 0.
8. Praktikan yang datang terlambat :
   - <= 30 menit : diperbolehkan mengikuti praktikum tanpa tambahan waktu Tes Awal
   - > 30 menit : tidak diperbolehkan mengikuti praktikum
9. Saat praktikum berlangsung, asisten praktikum dan praktikan:
   - Wajib menggunakan seragam sesuai aturan institusi.
   - Wajib mematikan/ mengkondisikan semua alat komunikasi.
   - Dilarang membuka aplikasi yang tidak berhubungan dengan praktikum yang berlangsung.
   - Dilarang mengubah pengaturan *software* maupun *hardware* komputer tanpa ijin.
   - Dilarang membawa makanan maupun minuman di ruang praktikum.
   - Dilarang memberikan jawaban ke praktikan lain.
   - Dilarang menyebarkan soal praktikum.
   - Dilarang membuang sampah di ruangan praktikum.
   - Wajib meletakkan alas kaki dengan rapi pada tempat yang telah disediakan.
10. Setiap praktikan dapat mengikuti praktikum susulan maksimal dua modul untuk satu mata kuliah praktikum.
    - Praktikan yang dapat mengikuti praktikum susulan hanyalah praktikan yang memenuhi syarat sesuai ketentuan institusi, yaitu: sakit (dibuktikan dengan surat keterangan medis), tugas dari institusi (dibuktikan dengan surat dinas atau dispensasi dari institusi), atau mendapat musibah atau kedukaan (menunjukkan surat keterangan dari orangtua/wali mahasiswa.)
    - Persyaratan untuk praktikum susulan diserahkan sesegera mungkin kepada asisten laboratorium untuk keperluan administrasi.
    - Praktikan yang diijinkan menjadi peserta praktikum susulan ditetapkan oleh Asman Lab dan Bengkel Informatika dan tidak dapat diganggu gugat.
11. Pelanggaran terhadap peraturan praktikum akan ditindak secara tegas secara berjenjang di lingkup Kelas, Laboratorium, Fakultas, hingga Universitas.

DAFTAR ISI

MODUL 1 RATIONAL ® SOFTWARE ARCHITECT

Tujuan Praktikum:

Mahasiswa memahami lingkungan (*environment*) dari kakas bantu (*tool*) yang digunakan

1.1 RATIONAL SOFTWARE ARCHITECT

Pada praktikum Rekayasa Perangkat Lunak: Teknik Berorientasi Objek (CSG3I1) ini, kakas (*tool*) yang digunakan adalah *Rational Software Architect v8.0* (RSA). Kakas ini merupakan salah satu produk dari IBM® Rational Software Delivery Platform, suatu solusi yang lengkap untuk mengembangkan perangkat lunak dan sistem berbasis-perangkat lunak (2007, 2007). Kebanyakan perkakas *platform* ini berbasis *Eclipse framework* yang merupakan kakas yang sebagai berikut:

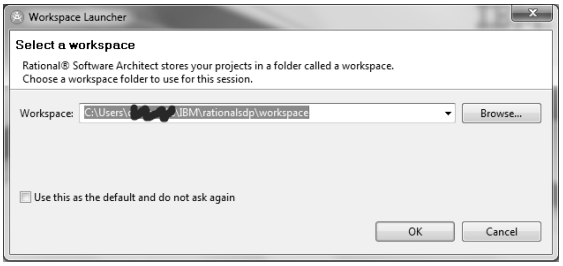
- Mempermudah kompleksitas lingkungan pengembangan dan mengkonsolidasikan penggunaan teknologi-teknologi pengembangan yang berbeda
- Membuat mudah pengembangan, pengujian, dan *deploy* perangkat lunak berkualitas tinggi dengan cara mengintegrasikan *enterprise environment*
- Memberikan dukungan multi-bahasa yang menyatukan teknologi Java™ 2 Platform Enterprise Edition (J2EE™), web services, UML, C++, dan teknologi-teknologi lainnya menjadi satu lingkungan pengembangan
- Memberikan fleksibilitas dan pilihan karena *open sources,* berjalan di banyak sistem operasi.

1.1.1 ECLIPSE

Eclipse adalah *platform* untuk membangun, men-*deploy,* dan mengatur perangkat lunak untuk semua *lifecycle* (2007, 2007). *Platform* Eclipse memberikan kemampuan untuk mengembangkan secara cepat perangkat lunak yang terintegrasi berbasiskan model *plug-in.* Antar muka pengguna Eclipse digunakan pada seluruh kakas IBM Rational Software Delivery Platform.

1.1.2 Workspace

Pada saat pertama kali membuka IBM Software Delivery Platform, dialog **Workspace Launcher** (Gambar 1-1~~Gambar 1-1~~) akan muncul. **Workspace** adalah lokasi untuk menyimpan hasil pekerjaan yang dilakukan dengan Eclipse. Setiap *resources* (*projects, folders,* dan *files*) yang dikerjakan akan masuk ke dalam *workspace* ini.

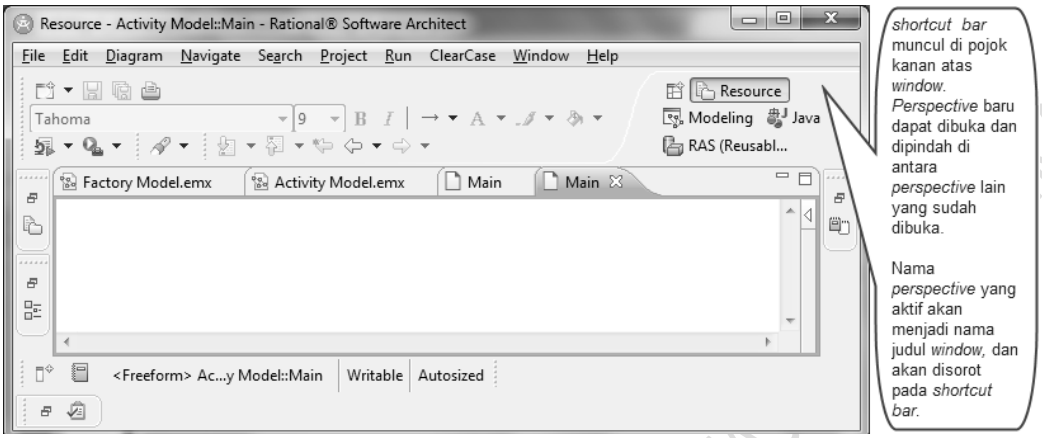


Gambar 1-1 *Workspace* adalah sebuah direktori, yang secara *default* akan dinamai 'workspace'

Worskpace bisa diletakkan dimana saja, *workspace* bisa saja berisi *project* baru sebagai *sub-directories,* atau hanya berupa *references* ke *projects* yang ada dimanapun pada *file system*. Jika memungkinkan, sebaiknya nama *path workspace* dibuat singkat. *Path* tersebut bisa saja sudah cukup panjang, dan Microsoft® Windows ® bisa bermasalah dengan nama *path* yang panjang.

1.1.3 WORKBENCH

Workbench bertindak sebagai sebuah *hub* untuk mengidentifikasi seluruh fungsi-fungsi *plug-in.* Masing-masing *workbench* memberikan satu atau beberapa *perspective* lebih, yang berisi *view* dan *editor* untuk menampilkan informasi. Dengan *workbench, resources* dapat dinavigasi, dan juga *workbench* dapat memperlihatkan dan mengubah isi dan *properties* dari *resources* ini



Gambar 1-2 *Workbench* dan *shortcut bar* pada RSA

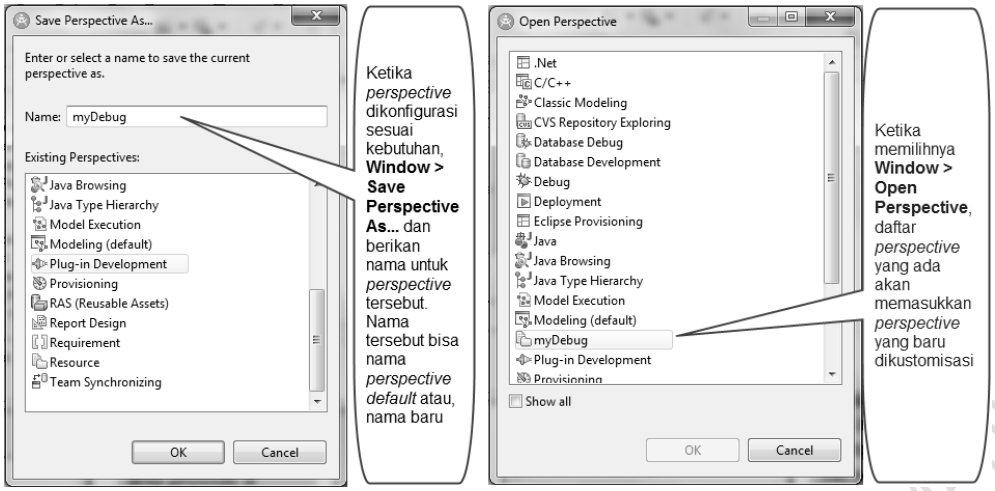
Saat *Welcome screen* tertutup, hal pertama yang akan terlihat adalah sebuah *dialog* dimana *workspace* dapat dipilih. *Workspace* adalah direktori tempat menyimpan hasil pekerjaan yang dilakukan. Untuk memilih direktori *default,* klik tombol OK. Pada awal dibuka, *window workbench* yang pertama kali muncul adalah *perspective Resource,* yang hanya menampilkan *view Welcome screen* (2007, 2007).

1.1.4 PERSPECTIVE

Perspective menentukan sekumpulan *editor* dan *view* yang disusun pada sebuah *layout* pertama untuk bermacam-macam *role* atau *task.* Misalkan, *layout* pertama dari *perspective* **Resource** menjadikan *view* dan *editor* **Navigator** terlihat, dan menyusunnya pada *layout default.* Beberapa *default view* berbeda untuk setiap *perspective*-nya. *Workbench* dapat dibuka menjadi beberapa *perspective,* atau sesuai dengan *perspective* yang ditentukan. *Perspective* yang berbeda lebih cocok untuk *task* yang berbeda. Beberapa *perspective* yang umum digunakan antara lain:

- Resource: *Perspective default* untuk kebanyakan kakas
- Java: *Perspective* untuk *projects* Java
- Java Browsing: *Perspective* untuk menjelajahi (*browsing*) struktur *projects* Java
- Debug: *Perspective* untuk melakukan *debugging* program
- Plug-in Development: *Perspective* untuk mengembangkan *plug-in* Eclpse

Untuk mengganti *perpective*: Window > Preferences > General > Perspectives. *Perspective* dapat dirancang dengan sekumpulan fungsi yang belum ditentukan namanya yang dapat dilakukan ketika di dalamnya. *Perspective* dapat dibuka lebih dari satu pada suatu waktu, atau dapat ditukar antar beberapa *perspective* dari *shortcut bar.* Ketika membuka *perspective,* sebenarnya satu *instance* dari *perspective* telah dibuka. Ketika dibuka, *instance* tersebut dapat disesuaikan seperti yang diinginkan, kemudian disimpan (*save*).

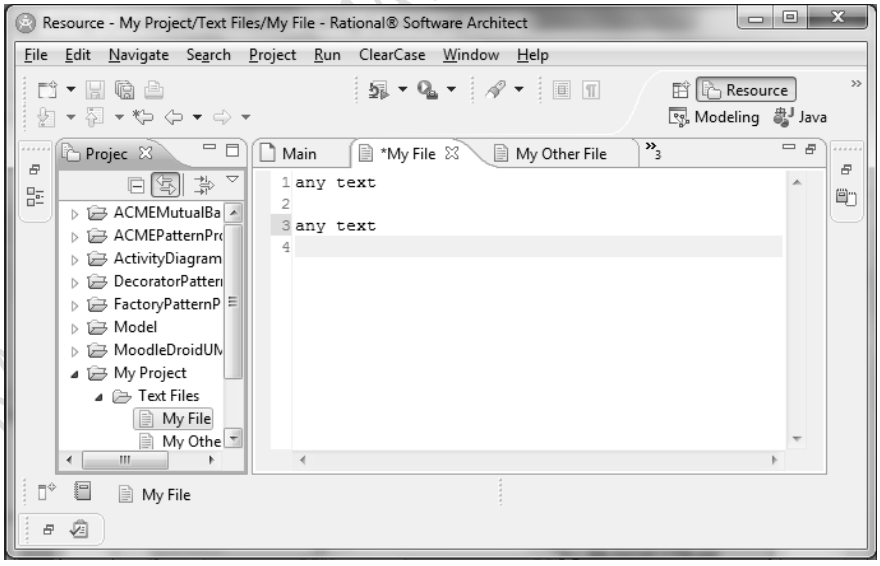


Gambar 1-3 Mengkonfigurasi *perspective* RSA

1.1.5 VIEWS DAN EDITORS

View digunakan untuk menavigasi hirarki informasi (seperti *resources* pada *workbench*), bukan *editor,* atau menampilkan *properties* untuk *editor* yang sedang aktif. Sedangkan *editor* digunakan untuk mengubah atau menjelajahi suatu *resource. View* dan *editor* adalah entitas visual utama yang muncul pada *Workbench.* Pada *perspective* tertentu, terdapat suatu area editor tunggal yang dapat berisi beberapa *editors,* dan beberapa *views.*

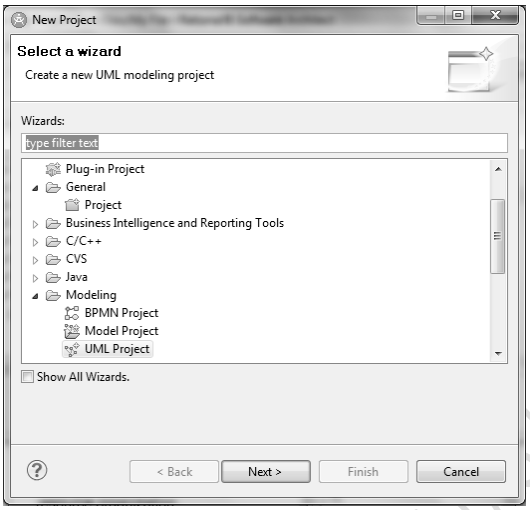
Editors memungkinkan pengubahan dokumen dari *workbench* untuk beberapa jenis artefaks, termasuk teks dan *diagram*. Tergantung pada tipe *file* yang diubah, *editor* yang tepat ditampilkan pada area *editor.* Misalkan, jika sebuah file txt sedang diedit, *editor* teks munsul pada area editor.



Gambar 1-4 Nama *file* muncul pada tab *editor.* Jika tanda asterisk (*) muncul di sisi kiri *tab,* ini menunjukkan bahwa *editor* memiliki perubahan yang belum disimpan

1.1.6 PROJECTS

Project adalah kumpulan sejumlah *files* dan *folders. Project* adalah wadah untuk mengatur *resources* lainnya yang berhubungan dengan *project* tersebut. *Projects* digunakan untuk membangun, manajemen versi, *sharing,* dan pengaturan *resource.*



Gambar 1-5 Membuat *project* baru

Seperti *folder, project* memetakan direktori pada *file system.* Ketika *project* dibuka, struktur *project* dapat diubah dan isinya dapat dilihat. Ketika *project* ditutup, perubahan tidak dapat dilakukan untuk *project* tersebut pada *workbench. Resources* dari *project* yang ditutup tidak akan muncul di *workbench,* akan tetapi *resources* akan terletak pada *file system* lokal. *Project* yang ditutup akan mengurangi jumlah *memory.* Karena *project* yang ditutup tersebut tidak diperiksa selama proses *builds,* menutup suatu *project* dapat meningkatkan waktu untuk *build.*

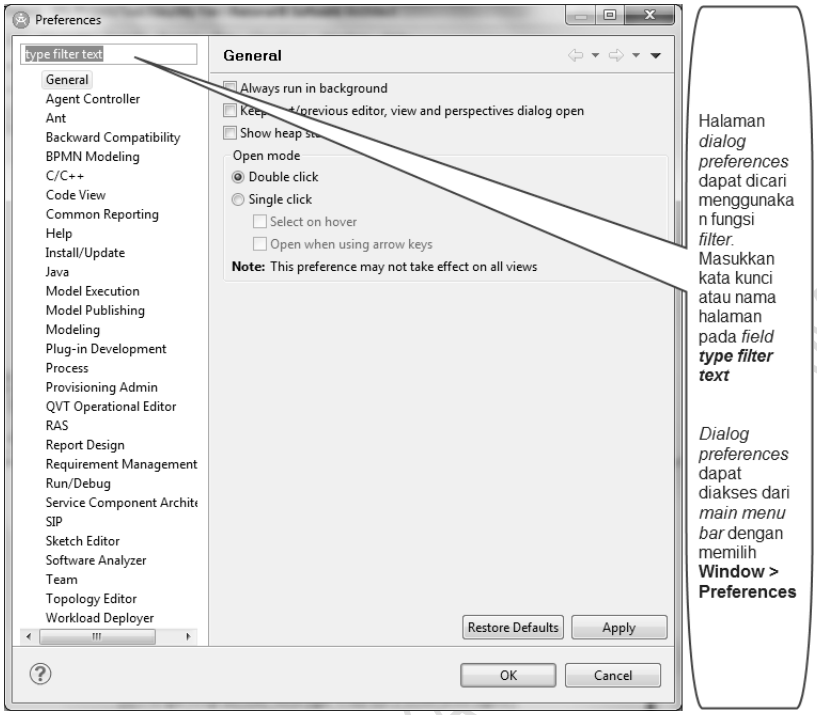
Untuk meng-*import project* yang ada pada *workspace* yang berbeda atau *project* yang sebelumnya telah ada pada satu *workpace,* kepada *Workbench,* **import wizard** dapat digunakan. Meng-*import* suatu *project* yang ada berarti memberikan akses untuk menjelajahi *file system* untuk *project* dan meng-*import*-nya seperti yang ditentukan saat itu. Artinya membuat *project* baru pada *workspace* yang dijalankan, tapi *project* yang sekarang digunakan sebenarnya tidak dipindahkan ke *file system, project* tersebut hanya diacu.

Hati-hati saat menghapus suatu *project* pada *workspace,* jika menghapus *project* dari *workspace* dan menekan tombol **Yes**, *files* para lokasi asli akan dihapus. Jika *projects* ingin dipindah pada direktori *workspace,* isinya harus di-*copy* ke direktori *workspace* secara manual, dan meng-*import*-nya dengan mem-*pointing* lokasi tersebut.

Ketika mengekspor sebuah *project,* beberapa format *file* dapat dipilih. Dua format yang umumnya digunakan adalah **.zip** (mengekspornya menjadi satu *file* tunggal) dan **File System** (mengekspornya menjadi berbasis *file system* pada hirarki yang terdefinisikan pada *Navigator Explorer* atau *Package Explorer*). Salah satu keuntungan utama melakukan ekspor adalah *file system* tersebut tidak harus di-*scan*, karena *project* itu tidak mungkin disimpan pada lokasi *default.* Ingat bahwa *project* yang diimpor tidak perlu terletak pada *workspace* yang dipilih.

1.1.7 PREFERENCES

Dialog Preferences digunakan untuk mengkustomisasi *settings* untuk *Workbench Eclipse,* dan juga *tools* dan komponen-komponennya.



Gambar 1-6 Dialog Preferences

1.1.8 TASKS

Dengan menggunakan *Tasks,* pekerjaan yang spesifik dapat ditelusuri (*track*) atau diberi pengingat (seperti *notes*). **View Tasks** menampilkan daftar *tasks* yang harus dilakukan, bersama dengan *references* ke *resources*-nya. *Tasks* dapat di-*generate* dari *templates,* atau dibuat secara manual. Untuk membuat *task* pada baris kode spesifik pada suatu *resource,* klik-kanan *editor's marker bar* untuk baris yang ingin dihubungkan dengan *task* tertentu, dan pilh **Add Task.**

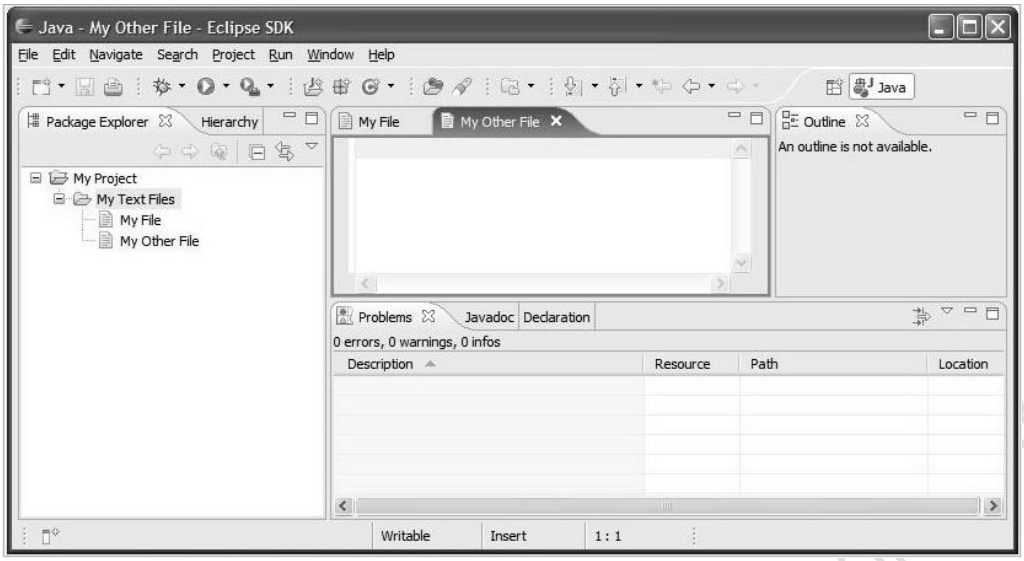
Berikut ini adalah contoh menambahkan *items* untuk *view Tasks* secara manual. Jika ingin menambahkan pengingat (*reminder*) untuk ditindaklanjuti di lain waktu, tapi *task* tersebut tidak ingin dihubungkan dengan *resources* tertentu, tambahkan *task* tersebut pada *view Tasks.* Karena *tasks* pada dasarnya adalah *to-do list, tasks* tidak hanya memberkan informasi *state* yang lengkap atau tidak lengkap, tapi juga tiga level prioritas untuk membantu mengatur *tasks.*

Gambar 1-7 Menambahkan *task* pada *resource editor*

1.2 LANGKAH-LANGKAH MEMBUAT ECLIPSE PROJECT

Untuk membuat *project* baru lakukan langkah-langkah di bawah ini:

1) Membuat *general project* baru
   a) Pindah *perspective* ke *Java Perspective,* klik **Window > Open Perspective > Java**
   b) Pada menu **File**, klik **New > Project**
   c) Pada *wizard* **New Project,** *expand* **General** kemudian pilih **Project.** Klik **Next**
   d) Beri nama *project* dengan 'My Project' dan kemudian tekan **Finish**
2) Membuat *folder* baru
   a) Klik-kanan 'My Project' pada ***Package Explorer*** dan pilih **New > Folder**
   b) Ketik 'Text Files' sebagai nama *folder,* dan klik **Finish**
3) Membuat *file* baru
   a) Pada *view* ***Package Explorer,*** klik-kanan *folder* yang baru saja dibuat dan pilih **New > File**
   b) Pastikan 'My Project/Text Files' dimasukkan sebagai *Parent Folder*
   c) Tuliskan 'My File' sebagai nama *file* dan klik **Finish**
4) Menyimpan pekerjaan
   a) Ketikkan sembarang teks pada *editor*
   b) Perhatikan tanda *asterisk* (*) di sebelah nama *file* pada *editor.*Tanda ini menunjukkan bahwa *file* telah diubah sejak terakhir disimpan
   c) Tekan CTRL + S untuk menyimpan pekerjaan. Perhatikan tanda *asterisk* yang sekarang sudah menghilang
5) Membuat *file* lainnya
   a) Klik **File > New > File**
   b) *Expand* 'My Project' dan pilih 'Text Files' sebagai *Parent Folder*
   c) Tuliskan 'My Other File' sebagai nama *file* dan klik **Finish**



Gambar 1-8 Tampilan *Package Explorer* yang seharusnya muncul

1.3 LANGKAH-LANGKAH MENGKUSTOMISASI PERSPECTIVE

Untuk mengkustomisasi *perspective,* lakukan langkah-langkah berikut:

1) Mengubah *Java Perspective*
   a) Pindah ke *Java Perspective* dengan cara, klik **Window > Open Perspective > Java**
   b) Klik **Window > Save Perspective As**
   c) Tuliskan 'My Perspective' pada *field Name,* klik **OK**
   d) Klik **Window > Open Perspective > Other** dan pastikan *perspective* baru sudah berhasil ditambahkan pada daftar yang muncul
2) Mengkustomisasi *perspective* baru
   a) Klik **Window > Customize Perspective** untuk menampilkan *dialog* **Customize Perspective**
   b) Pilih **New** pada *context box* 'Submenus'
   c) Hapus kategori **Java and Team**, dan pastikan kategori **General** terpilih
   d) Klik *tab* **Commands** untuk melihat perintah-perintah apa saja yang ada pada *perspective* yang terkustomisasi
   e) Pilih kategori **CVS** dari *pane* sebelah kiri dari perintah-perintah yang ada, dan klik **OK**.
   f) Pastikan *toolbar* **Checkout** dari tombol **CVS** telah ditambahkan pada *toolbar,* dan juga pastikan **File > Now** memasukkan hanya opsi-opsi yang telah ditentukan.
3) Menggunakan *windows* Eclipes ganda
   a) Pilih **Select > New Window** untuk membuka *window* Eclipse baru. Aksi ini akan membuka dua *windows* pada *workspace* yang sama. Ubah ukuran dan posisi kedua *windows* tersebut sehingga keduanya dapat terlihat berbarengan.
   b) Gunakan *perspective* yang telah dikustomisasi sebelumnya untuk membuat *general project* yang baru
   c) Namakan *project* baru tersebut dengan 'My Other Project' dan klik **Next**. Pastikan *project* tersebut berhasil ditambahkan pada kedua *windows*.
   d) Buatlah sebuah *file* dengan nama 'My Other File' pada *project* tersebut.
   e) Buka satu *edior* pada 'My Other File' di masing-masing dua *windows* tersebut dan tuliskan sembarang teks pada salah satunya. Periksa kedua *windows* terbarui secara serempak.
4) Menggunakan *working sets*
   a) Klik menu yang ada di bagian atas sebelah kanan dari *view Package Explorer* dan pilih **'Select Working Set..'**

b) Pada *dialog Working Sets,* klik '**New**..' untuk membuka *dialog New Working Set.* Pilih '**Resource'** sebagai tipe *Working Set* dan klik '**Next**'.
c) Tuliskan 'My Project' sebagai nama dari *Working Set* dan pilih 'My Project' sebagai isi *Working Set.* Klik '**Finish**' untuk membuat *working set* dan kembali ke *dialog Working Set.*
d) Pilih opsi '**Selected Working Sets**', pilih kotak 'My Project' sebagai *Working Set* baru, dan klik **OK** untuk mengaktifkannya bagi *window* tersebut. Perhatikan hanya 'My Project' saja yang muncul sekarang pada *view Package Explorer*.
e) Pindah ke *window* lainnya, buat sebuah *Working Set* baru namakan 'My Other Project' dan aktifkan *Working Set* itu untuk *window* tersebut. Lihat sekarang terdapat dua *window* yang masing-masing menampilkan isi dari 'My Project' dan lainnya menampilkan 'My Other Project'.
f) Tutup *window* kedua dengan meng-klik ikon *close* pada pojok kanan *window.*
g) Klik *menu* tersebut pada bagian kanan atas dari *view Package Explorer* dan pilih **'Deselect Working Set'**.

MODUL 2 USE CASE DIAGRAM

Tujuan Praktikum:

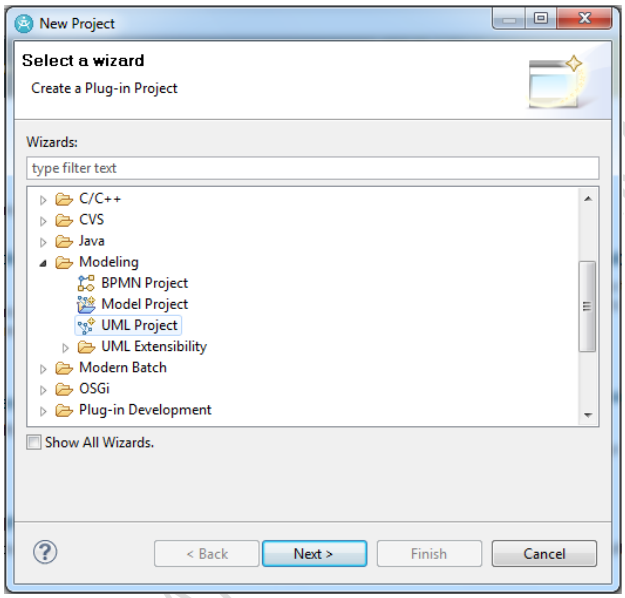
1. Praktikan mampu membuat sebuah scenario suatu system yang nantinya dapat diimplementasikan menjadi sebuah perangkat lunak
2. Praktikan bisa memahami alur dari setiap tahap yang digunakan dalam perancangan perangkat lunak menggunakan UML
3. Praktikan dapat memahami hubungan antara *actor* dengan *use case diagram*
4. Praktikan mampu membuat *use case diagram* dari skenario yang telah ada
5. Praktikan dapat menentukan *candidat eclass* dari skenario yang telah ada

2.1 MEMBUAT PROJECT BARU

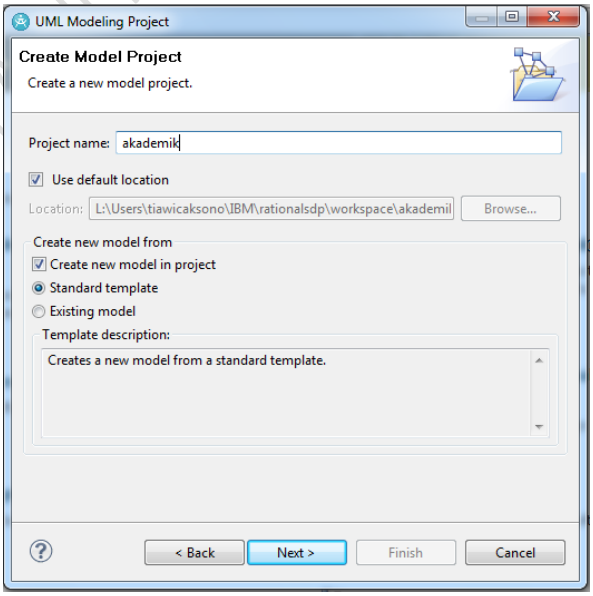
Sebelum membuat use case diagram, kita mulai dulu dengan membuat project.

Klik File > New > Project >Modeling > UML Project > klik next. Kemudian Isikan nama project > klik next.

*pada bagian use default location dapat di uncentang untuk menentukan lokasi penyimpanan.

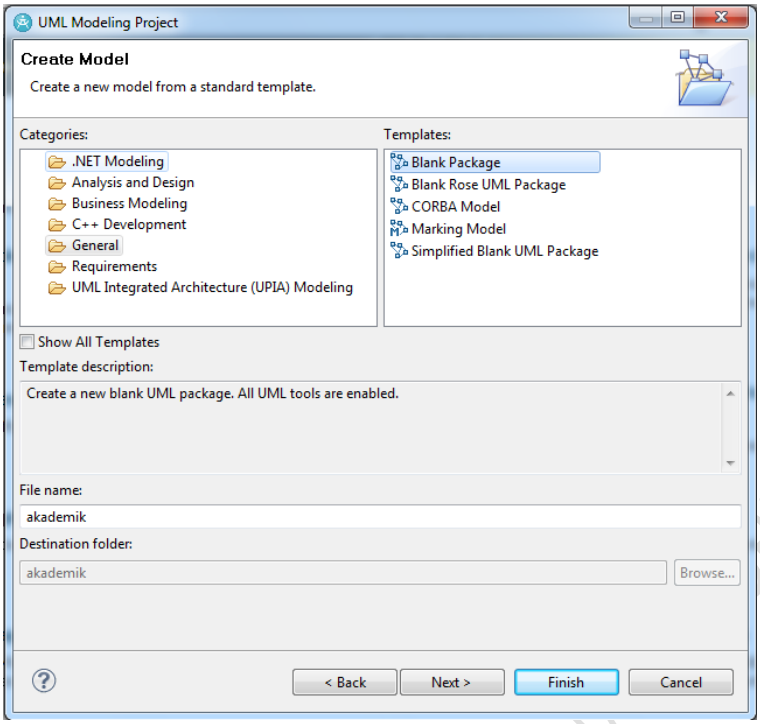


Gambar 2-1 Memilih wizard



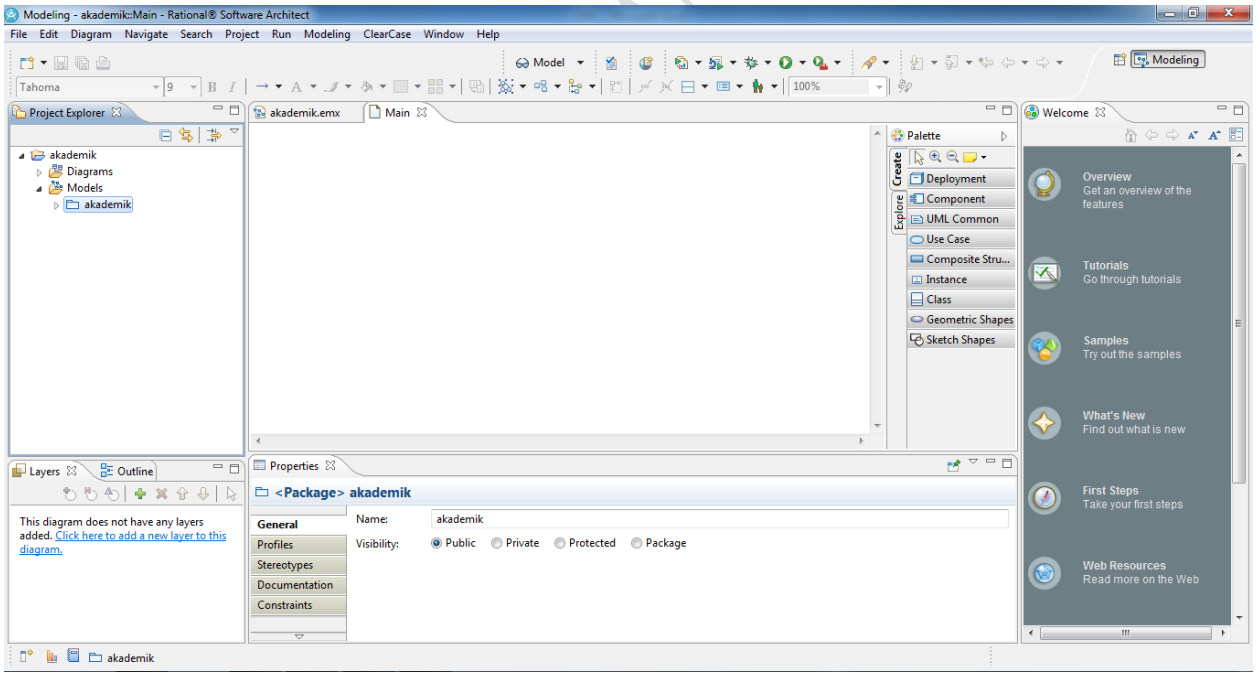
Gambar 2-2 Membuat model project

Pada bagian template: pilih blank package Pada file name: isikan nama file **Klik Next > Klik Finish**



Gambar 2-3 Membuat model

Hasil pembuatan project

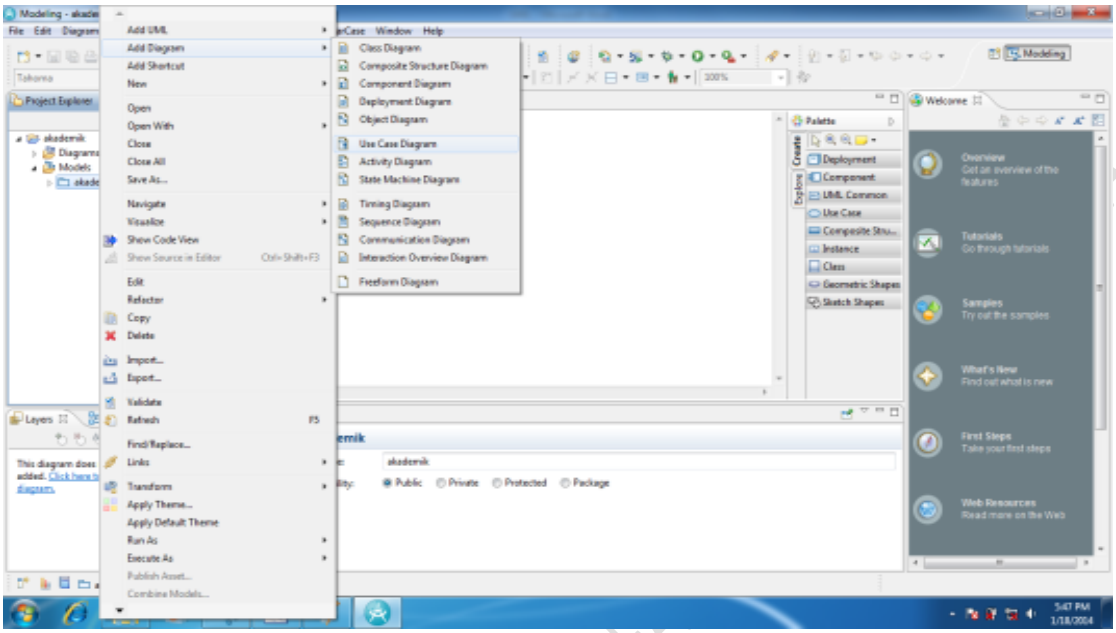


Gambar 2-4 Hasil pembuatan model

2.2 LANGKAH – LANGKAH MEMBUAT USE CASE DIAGRAM

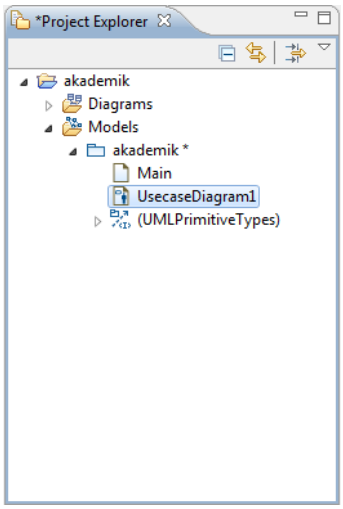
Berikut merupakan langkah-langkah dalam membuat usecase diagram :

Klik kanan pada file yang telah dibuat (misal: akademik) > Add Diagram > klik Use Case Diagram

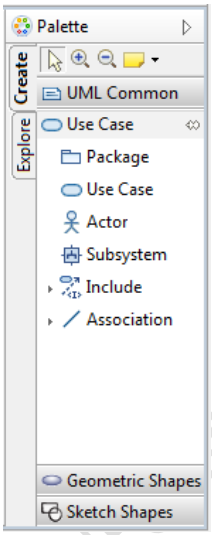


Gambar 2-5 Langkah membuat usecase diagram

Setelah itu akan tampil halaman seperti di bawah ini



Gambar 2-6 hasil pembuatan menu usecase diagram



Gambar 2-7 menu yang digunakan usecase diagram

2.2.1 USECASE

Use case adalah gambaran fungsionalitas dari suatu sistem, sehingga *customer* atau pengguna sistem paham dan mengerti mengenai kegunaan sistem yang akan dibangun.

Sebuah use case merupakan sekumpulan urutan dari aksi atau langkah-langkah, termasuk variant-nya (skenario lain yang mungkin terjadi), yang dilakukan oleh sistem untuk memberikan hasil yang dapat diamati dan diukur oleh user. Secara grafis use case dilambangkan dengan bentuk elips.

Sebuah use case hendaknya spesifik namun fungsionalitas yang dilakukan juga tidak boleh terlalu kecil. Untuk menamai sebuah use case dapat menggunakan kata kerja aktif yang menggambarkan apa yang dilakukan oleh use case tersebut.

Use Case dibagi menjadi dua kategori, yaitu:

1. Use case konkrit: Use case yang dibuat sesuai kebutuhan actor
2. Use case abstrak: Use case yang tidak bisa berdiri sendiri

Cara menentukan Use Case dalam suatu sistem:

- Pola perilaku perangkat lunak aplikasi.
- Gambaran tugas dari sebuah *actor*.
- Sistem atau “benda” yang memberikan sesuatu yang bernilai kepada *actor*.
- Apa yang dikerjakan oleh suatu perangkat lunak (bukan bagaimana cara mengerjakannya).

use case

Catatan : Use case diagram adalah penggambaran sistem dari sudut pandang pengguna sistem tersebut / external view (user), sehingga pembuatan use case lebih dititikberatkan pada fungsionalitas yang ada pada sistem, apa yang dilakukan, dan bukan bagaimana proses itu dilakukan.

2.2.2 AKTOR

Untuk dapat terciptanya suatu *use case diagram* diperlukan beberapa *actor.* Dalam context use case, user dari suatu use case disebut aktor. Seorang aktor merupakan sekumpulan peran yang berkaitan yang dimainkan pada saat berinteraksi dengan use case yang bersangkutan. Biasanya seorang aktor mewakili peran yang dimainkan oleh manusia, perangkat keras, atau bahkan sistem lain.

Seorang aktor dapat melakukan banyak use case, sebaliknya sebuah use case dapat dilakukan oleh banyak aktor. Selain itu seorang user dapat berperan menjadi lebih dari satu aktor.

Aktor hanya dapat berhubungan dengan use case dalam bentuk asosiasi. Sebuah asosiasi antara aktor dan use case berarti bahwa aktor dan use case saling berkomunikasi dan masing masing memungkinkan untuk mengirim dan menerima pesan. Aktor hanya berinteraksi dengan use case, tetapi tidak memiliki kontrol akan use case tersebut. Secara grafis seorang aktor dilambangkan dengan *stick man.*

Sebuah *actor* mungkin hanya memberikan informasi inputan pada sistem, hanya menerima informasi dari sistem atau keduanya menerima, dan memberi informasi pada sistem. *Actor* dapat digambarkan secara secara umum atau spesifik. Untuk membedakannya kita dapat menggunakan *relationship.*

Ada beberapa kemungkinan yang menyebabkan *actor* tersebut terkait dengan sistem antara lain:

- Yang berkepentingan terhadap sistem dimana adanya arus informasi baik yang diterimanya maupun yang dia inputkan ke sistem.
- Orang ataupun pihak yang akan mengelola sistem tersebut.
- *External resource* yang digunakan oleh sistem.
- Sistem lain yang berinteraksi dengan sistem yang akan dibuat.

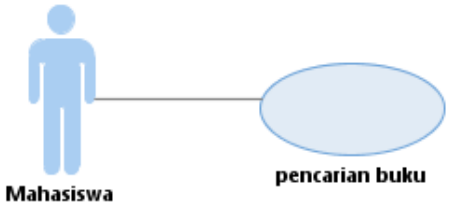
Jadi, hanya yang berinteraksi dengan sistem saja yang akan dianggap sebagai aktor. Sementara orang ataupun sesuatu yang tidak berinteraksi dengan sistem tidak bisa dianggap sebagai aktor.

Catatan : ada beberapa yang berpendapatan bahwa asosiasi antara aktor dan use case tidak melambangkan alur pengiriman pesan, namun lebih kepada inisiator dari use case tersebut. Inti dari dua pendapat ini sama, yaitu interaksi dan peran user terhadap suatu use case.

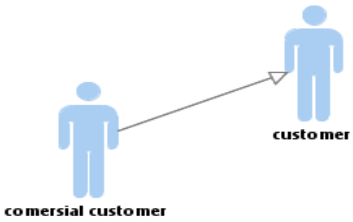
2.2.3 RELASI

Ada beberapa relasi yang terdapat pada *use case diagram*:

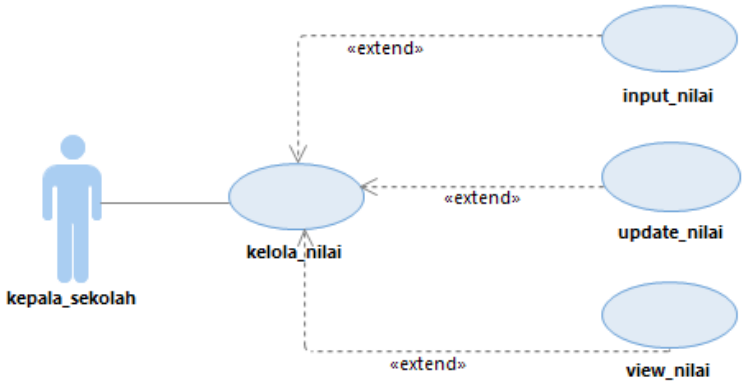
1. ***Association***, Menghubungkan elemen dengan proses pertukaran informasi. Dilambangkan dengan garis tegas tanpa panah. Contoh :



2. ***Generalization***, Generalisasi disini berarti inheritance (pewarisan), dimana sebuah elemen (use case atau aktor) dapat merupakan spesialisasi dari elemen lainnya. Dilambangkan dengan garis tegas yang memiliki panah tertutup.

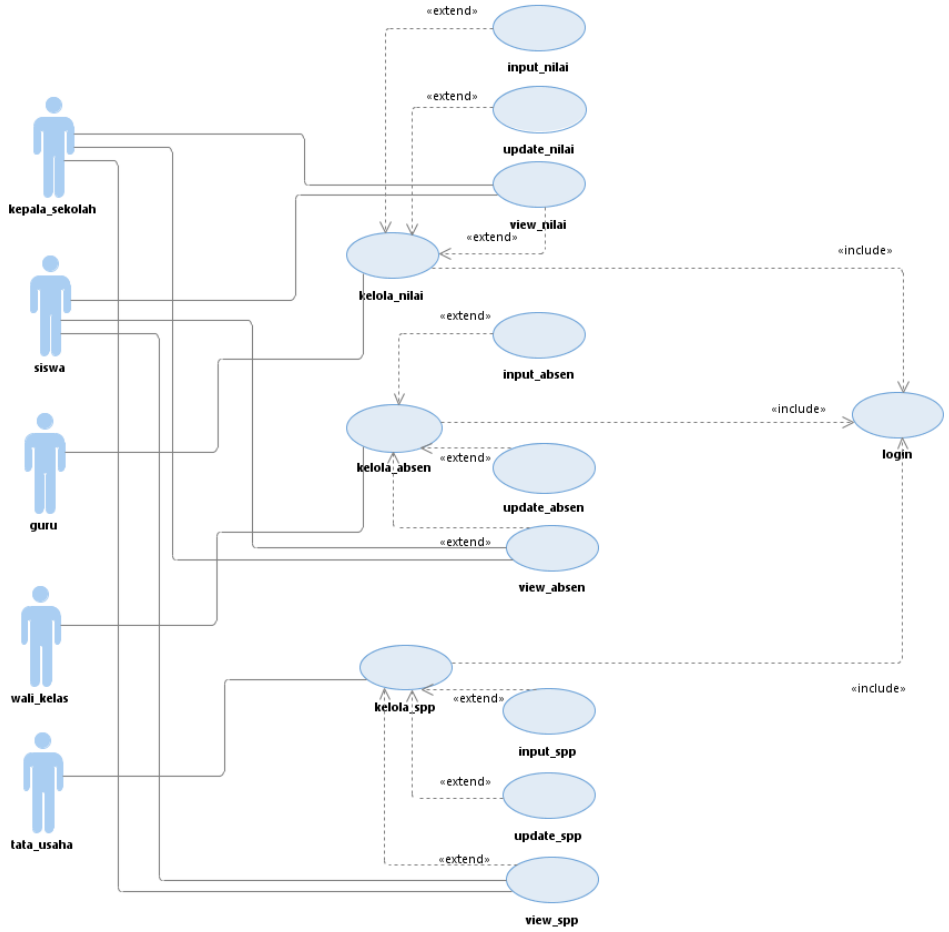


3. ***Aggregation***, bentuk assosiation dimana sebuah elemen berisi elemen lainnya.



4. *Dependency*,merupakan ketergantungan elemen terhadap elemen lain, dependency dalam diagram use case secara umum memiliki tiga stereotype yang dilambangkan dengan garis putus-putus, antara lain:
    1. **<<include>>**, yaitu kelakuan yang harus terpenuhi agar sebuah *event* dapat terjadi, dimana pada kondisi ini sebuah *use case* adalah bagian dari *use case* lainnya. Atau dengan kata laen, use case yang harus dilakukan sebelum melakukan use case yang di include.
    2. **<<extends>>**, kelakuan yang hanya berjalan di bawah kondisi tertentu seperti menggerakkan alarm. Jadi sifatnya kondisional, mungkin dilakukan dan mungkin tidak.
    3. <<**communicates**>>, mungkin ditambahkan untuk asosiasi yang menunjukkan asosiasinya adalah *communicates association* . Ini merupakan pilihan selama asosiasi hanya tipe *relationship* yang dibolehkan antara *actor* dan *use case*.

Contoh Usecase Diagram Akademik:



Gambar 2-8 Contoh usecase diagram

1.3 SKENARIO USECASE

Nomor : HLUC-01
Nama use case : kelola nilai
Actor : guru
Type : Primary
Tujuan : Melakukan proses kelola nilai
Deskripsi : Dengan memilih menu ini guru dapat mengeloh nilai siswa mereka , adapun kelola nilai itu terdiri dari kegiatan insert, update, delete nilai.

Actor	Sistem
1. Jika ingin mengelolah nilai, guru dapat memilih menu nilai pada sistem tentunya dengan login terlebih dahulu.	
	2. sistem akan memeriksa validitas data yang dimasukkan, apabila valid maka dapat mengakses sistem, kemudian dapat menggunakannya
	3. sistem akan menyimpan data kelola yang diinputkan.

Nomor : GLUC-01
Nama use case : View absen
Actors : kepala sekolah
Type : Secondary
Tujuan : User dapat melihat absen
Deskripsi : Menu ini memudahkan user untuk melihat presensi kehadiran.

Actor	Sistem
1. User memilih view absen	
	2. sistem akan menampilkan presensi yang disimpan pada database.

MODUL 3 CLASS DIAGRAM

Tujuan Praktikum::

1. Mahasiswa memahami kembali dasar-dasar class diagram
2. Mampu membuat Class Diagram dengan studikasus yang disediakan

3.1 PENGERTIAN OBJECT

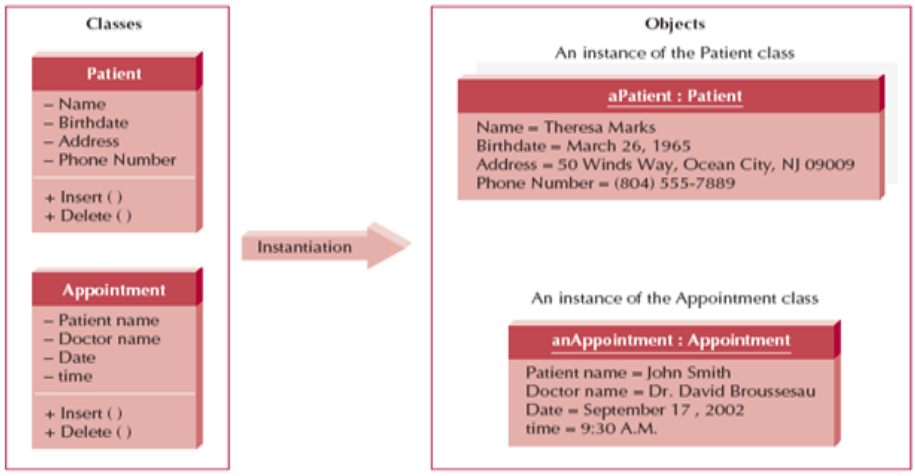
Object adalah gambaran nyata dari sebuah entitas baik dunia nyata atau konsep dengan batasan-batasan dan pengertian yang tepat. *Object* bisa mewakili sesuatu yang nyata seperti computer, mobil atau dapat berupa konsep seperti proses kimia, transaksi bank, permintaan pembelian dan sebagainya. Setiap object dalam system memiliki tiga karakteristik yaitu :

- **State,** merupakan satu kondisi / keadaan dari object yang mungkin ada. Status dari object akan berubah setiap waktu dan ditentukan oleh sejumlah property dan relasi dengan object lainnya.
- **Behavior (sifat)** menentukan bagaimana object merespon permintaan dari object lain dan melambangkan setiap hal yang dapat dilakukan. Sifat ini diimplementasikan dengan sejumlah operasi untuk object.
- **Identity (identitas)** artinya setiap object yang ada dalam suatu system adalah "unik".

Pada UML, sebuah object digambarkan dengan segiempat dan nama dari object diberi garis bawah.

rectA : Rectangle

Class adalah deskripsi sekumpulan *object* yang memiliki kesamaan atribut, operasi, *relationship* dan *semantics*. Dengan kata lain, sebuah *class* merupakan *blueprint* / *template* / cetakan dari satu atau lebih *object*. Sebuah *class* dapat menciptakan (instansiasi) satu atau lebih *object,* tetapi sebuah *object* hanya dapat mereferensi pada satu *class* saja.



Gambar 3-1 Perbedaan classes dan objects

3.2 CLASS DIAGRAM

Class diagram dapat membantu dalam memvisualisasikan struktur kelas-kelas dari suatu sistem dan merupakan tipe diagram yang paling di temui dalam pemodelan sistem berbasis *object-oriented*. Class Diagram memperlihatkan sekumpulan *class, interface*, dan *collaborations* dan relasi yang ada didalamnya. Selama proses analisa, class diagram memperhatikan aturan-aturan dan tanggung jawab entitas yang menentukan perilaku sistem. Selama tahap desain, class diagram berperan dalam menangkap struktur dari semuaclass yang membentuk arsitektur sistem yang dibuat. Kita memodelkan *class diagram* untuk memodelkan *static design view* dari suatu sistem.

Class diagram dapat digunakan untuk :

1. Memodelkan *vocabulary* dari suatu sistem
2. Menggambarkan kolaborasi sederhana
3. Memodelkan *logical database schema*

3.2.1 CLASS

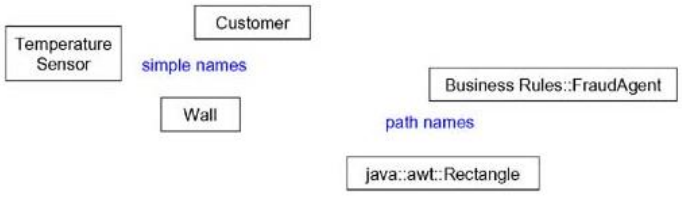
Pada UML, class digambarkan dengan segi empat yang dibagi menjadi 3 bagian. Bagian yang pertama merupakan nama dari kelas. Bagian yang tengah merupakan struktur dari class (atribut) dan bagian ketiga merupakan sifat / *behavior* /operasi dari class tersebut.

Gambar 3-2 Penggambaran class pada UML (Booch, Rumbaugh, & Jacobson, 2000)

- **Nama kelas**

  Setiap class harus memiliki sebuah nama yang dapat digunakan untuk membedakannya dariclass lain. Penamaan class menggunakan kata benda tunggal yang merupakan abstraksi yang terbaik.

  Namaclass dapat dituliskan dengan 2(dua) cara : 1) hanya menuliskan nama dariclass (*simple name*)dan 2) namaclass diberi *prefix* nama *package* letak class tersebut (*path name*).



Gambar 3-3 Penamaan Class

  Penulisan nama kelas, huruf pertama dari setiap kata pada namaclass ditulis dengan menggunakan huruf kapital. Contohnya, Customer dan FraudAgent.

- **Atribut**

  Attribute adalah salah salah satu *property* yang dimiliki oleh class yang menggambarkan batasan dari nilai yang dapat dimiliki oleh *property* tersebut. Sebuah class mugkin memiliki beberapa *attribute* atau tidak sama sekali. Sebah atribut merepresentasikan beberapa *property* dari sesuatu yang kita modelkan, yang dibagi dengan semua *object* dari semua *class* yang ada. Contohnya, setiap tembok memiliki tinggi, lebar dan ketebalan.

Atribut dalam implementasinya akan digambarkan sebagai sebuah daftar (list) yang diletakkan pada kompartemen kedua (di bawah nama kelas).

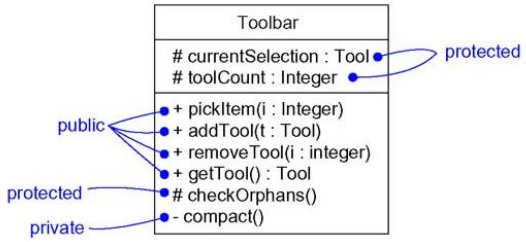
Untuk penulisan atribut kelas, biasanya huruf pertama dari tiap kata merupakan huruf kapital, kecuali untuk huruf awal. Contoh : birthDate, length.

- **Operasi**
  Sebuah operasi adalah sebuah implementasi dari layanan yang dapat diminta dari beberapa *object* dan *class,* yang mempengaruhi *behavior*. Dengan kata lain operasi adalah abstraksi dari segala sesuatu yang dapat kita lakukan pada sebuah *object* dan ia berlaku untuk semua *object* yang terdapat dalam *class* tersebut. *Class* mungkin memiliki beberapa operasi atau tanpa operasi sama sekali. Contohnya adalah sebuah *class* Kotak dapat dipindahkan, diperbesar atau diperkecil. Biasanya, pemanggilan operasi pada sebuah object akan mengubah data atau kondisi dari *object* tersebut. Operasi ini dalam implementasinya digambarkan di bawah atribut dari sebuah *class.*

- **Visibility**
  Visibility merupakan property yang sangat penting dalam pendefinisian atribut dan operasi pada suatu *class.Visibility* menspesifikasikan apakah atribut/operasi tersebut dapat digunakan/diakses oleh *class* lain. UML menyediakan 3 buah tingkat *visibility*, yaitu:

Visibility	Keterangan
public (+)	Dapat diakses oleh *class* lain. Dilambangkan dengan tanda
protected (#)	Hanya dapat diakses oleh *class* itu sendiri dan *class* turunannya (*sub class*)
private (-)	Hanya dapat diakses oleh *class* itu sendiri.

Tabel 3-1 Tingkat visibility



Gambar 3-4 Visibility pada operasi

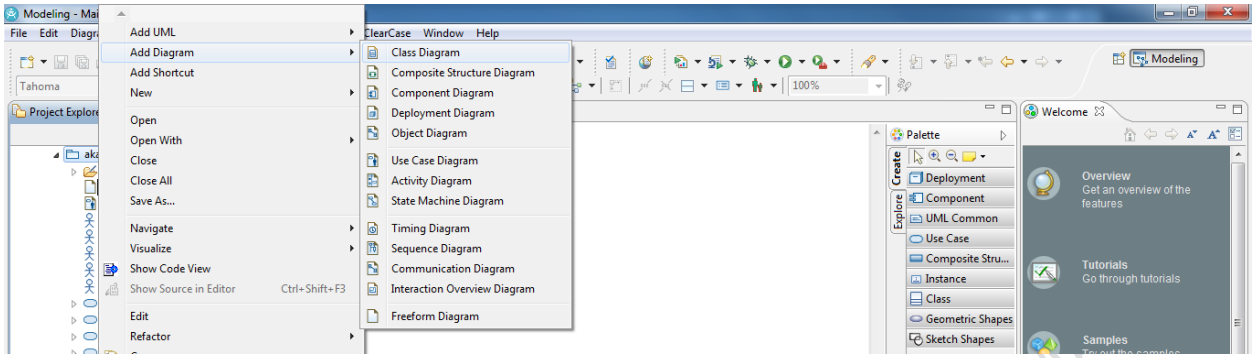
Pengorganisasian atribut dan operasi

Ketika menggambarkan sebuah class kita tidak perlu menampilkan seluruh atribut atau operasi. Karena dalam sebagian besar kasus kita tidak dapat menampilkannya dalam sebuah gambar, karena terlalu banyaknya atribut atau operasinya, bahkan terkadang tidak perlu karena kurang relevannya atribut atau operasi yang akan ditampilkan. Sehingga kita dapat menampilkan atribut dan operasinya hanya sebagian atau tidak sama sekali. Kosongnya tempat pengisian bukan berarti tidak ada. Karena itu kita dapat menambahkan tanda ellipsis (…) pada akhir daftar yang menunjukkan bahwa masih ada atribut atau operasi yang lain.

3.3 LANGKAH – LANGKAH MEMBUAT CLASS DIAGRAM

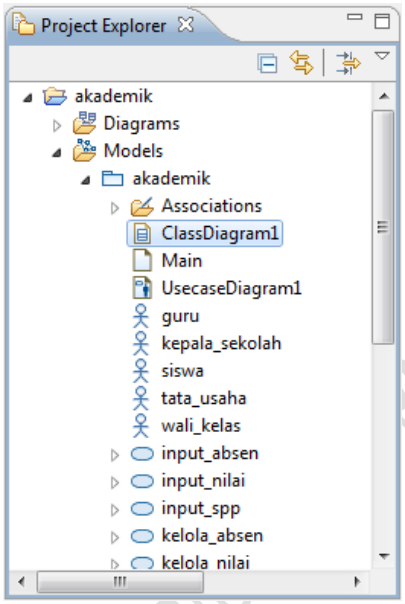
Berikut merupakan langkah-langkah dalam membuat class diagram :

Klik kanan pada file yang telah dibuat (misal: akademik) > Add Diagram > klik Use Case Diagram

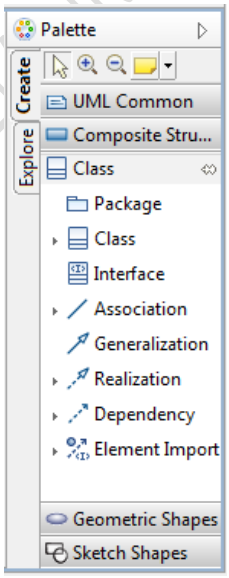


Gambar 3-5 Menu pilihan membuat usecase

Setelah itu akan tampil halaman seperti di bawah ini



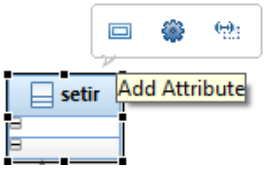
Gambar 3-6 Project explorer usecase



Gambar 3-7 Komponen usecase pada RSA

Jika ingin menambah atribut pada class klik simbol persegi pada bagian atas class

Jika ingin menambah operation tekan klik icon roda pada atas class



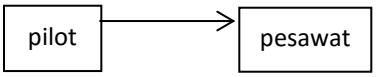
Gambar 3-8 add atribute

3.4 RELASI ANTAR CLASS

Relasi atau *relationship* menghubungkan beberapa objek sehingga memungkinkan terjadinya interaksi dan kolaborasi diantara objek-objek yang terhubung. Dalam pemodelan *class diagram*, terdapat tiga buah relasi utama yaitu *association, dependency* dan *generalization.*

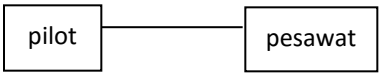
1. Association

Relasi asosiasi merupakan relasi structural yang menspesifikasikan bahwa satu objek terhubung dengan objek lainnya. Relasi ini tidak menggambarkan aliran data, sebagaimana yang terdapat pada pemodelan desain pada analisa terstruktur. Relasi asosiasi dapat dibagi menjadi 2(dua) jenis, yaitu *uni-directional association* dan *bidirectional association. Uni-directional association* digambarkan dengan (⟶). Tanda arah panah menunjukkan bahwa hanya terdapat hubungan satu arah.



Pada contoh diatas, terlihat objek pilot memiliki *uni-directional association* dengan objek pesawat. Relasi *uni-directional* diatas memungkinkan objek pilot untuk memanggil *property* dari objek pesawat. Namun tidak berlaku sebaliknya. Objek pesawat tidak dapat mengakses *property* dari objek pilot.

Sedangkan untuk *bi-directional association* memungkinkan terjadinya hubungan dua arah. Dalam *class diagram,* relasi *bi-directional* digambarkan dengan (———).



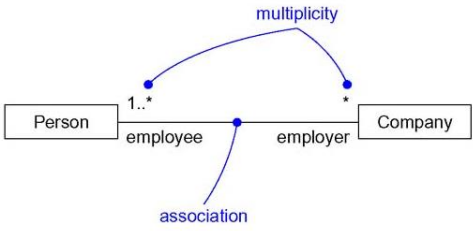
Berbeda dengan contoh sebelumnya, pada gambar diatas, objek pilot dapat memanggil *property* yang dimiliki oleh objek pesawat. Begitu juga sebaliknya, objek pesawat juga dapat memanggil *property* dari objek pilot.

2. Multiplicity

Multiplicity menentukan/mendefinisikan banyaknya *object* yang terhubung dalam suatu relasi. Indikator *multiplicity* terdapat pada masing-masing akhir garis relasi, baik pada asosiasi maupun agregasi. Beberapa contoh *multiplicity* adalah:

Potencial Multiplicity Values	
Indicator	**Meaning**
0..1	Zero or one
1	One only
0..*	Zero or more
1..*	One or more
N	Only *n* (where *n* > 1)
0..n	Zero to *n* (where *n* > 1)
1..n	One to *n* (where *n* > 1)

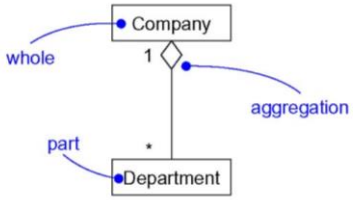
Tabel 3-2 Potencial multiplicity values



Gambar 3-9 Multiplicity pada asosiasi

3. Aggregation

Aggregation relationship adalah bentuk khusus dari asosiasi dimana induk terhubung dengan bagian-bagiannya. *Aggregation* merepresentasikan relasi *"has-a"*, artinya sebuah *class* memiliki/terdiri dari bagian-bagian yang lebih kecil. Dalam UML, relasi agregasi digambarkan dengan *open diamond* pada sisi yang menyatakan induk (whole).



Gambar 3-10 "whole-part" pada relasi agregasi

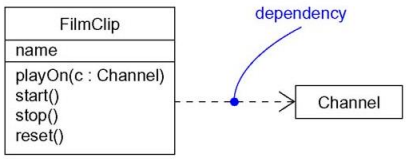
Pertanyaan-pertanyaan di bawah ini dapat digunakan untuk menentukan apakah asosiasi seharusnya menjadi agregasi:

- Apakah klausa *has-a* (bagian dari) digunakan untuk menggambarkan relasi?
- Apakah beberapa operasi di induk secara otomatis dapat dipakai pada bagian-bagiannya? Sebagai contoh, penghapusan sebuah *company*, maka akan menghapus *departement*-nya

4. Dependency

Dependency merupakan sebuah relasi yang menyebutkan bahwa perubahan pada satu *class* (misal *class* event), maka akan mempengaruhi *class* lain yang menggunakannya (misal *class* window), tetapi tidak berlaku sebaliknya.

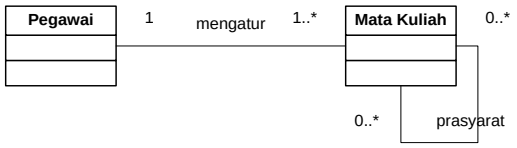
Pada umumnya, relasi *dependency* dalam konteks *Class* Diagram, digunakan apabila terdapat satu *class* yang menggunakan / meng-*instance class* lain sebagai argumen dari sebuah method. Perhatikan contoh di bawah, bila spesifikasi dari *class* Channel berubah, maka method playOn pada *class* FilmClip juga akan berubah.



Gambar 3-11Relasi *Dependency*

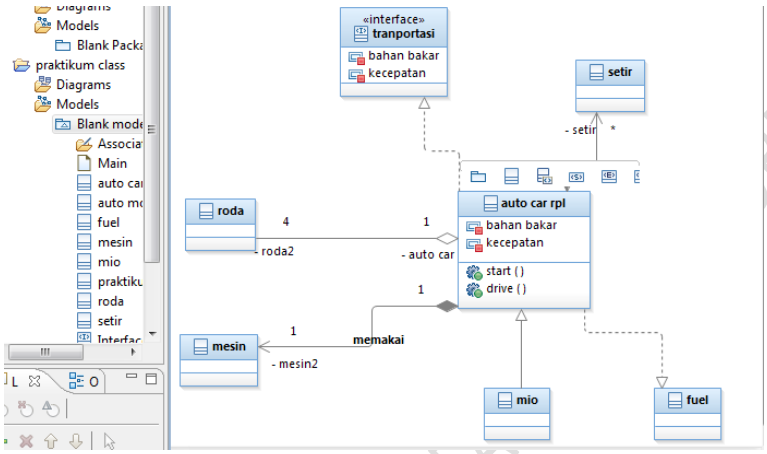
5. Reflexive Relationship

Multiple object pada *class* yang sama dapat saling berkomunikasi satu dengan yang lainnya. Hal ini ditunjukkan pada *class diagram* sebagai *reflexive association* atau *aggregation*. Penamaan *role* lebih disukai untuk digunakan pada *reflexive relationships* daripada penamaan *association relationship*.



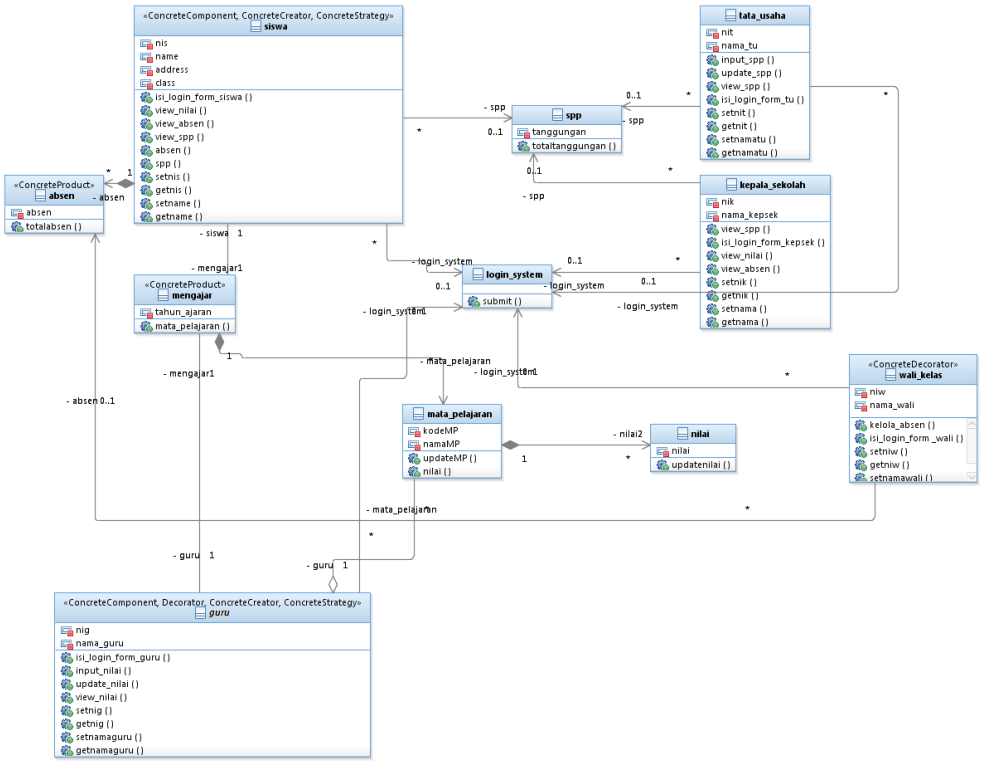
Gambar 3-12 Reflexive relationship

Contoh tampilan class diagram autocar rpl:



Gambar 3-13 Contoh diagram autocar rpl

Contoh Class Diagram Akademik:



Gambar 3-14 Contoh diagram akademik

MODUL 4 SEQUENCE DIAGRAM & COMMUNICATION DIAGRAM

Tujuan Praktikum::

Mahasiswa mampu membuat sequence diagram dan communication diagram dengan studi kasus yang disediakan

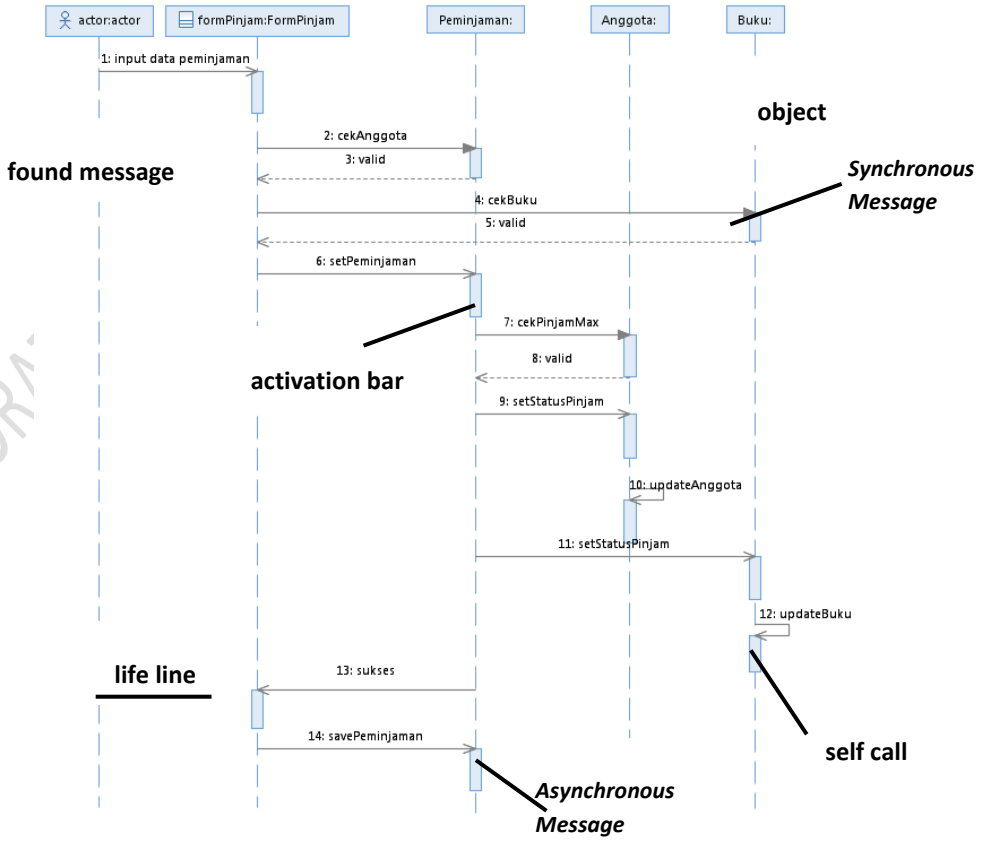
4.1 SQUENCE DIAGRAM

Sequence diagram menggambarkan interaksi antara sejumlah object dalam urutan waktu. Umumnya sebuah sequence diagram menangkap behavior dari suatu skenario (best case). Diagram ini menunjukkan sejumlah object dan pesan-pesan yang dilewatkan antara object-object ini dalam skenario tersebut.

Dalam *diagram sequence*, setiap *object* hanya memiliki garis yang digambarkan garis putus-putus ke bawah. Pesan antar *object* digambarkan dengan anak panah dari *object* yang mengirimkan pesan ke *object* yang menerima pesan.

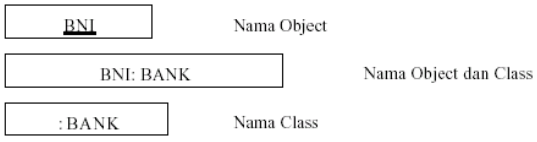
4.1.1 KOMPONEN-KOMPONEN PADA SEQUENCE DIAGRAM

Kalimat pembuka/penghubung ke gambar?



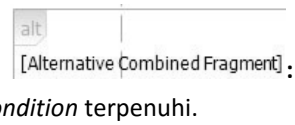
Gambar 4-1 Komponen pada sequence diagram

- ***Object*** didapat dari class diagram. Dalam UML, *object* pada *diagram sequence* digambarkan dengan segi empat yang berisi nama dari *object* yang digarisbawahi. Pada *object* terdapat 3 cara untuk menamainya yaitu : nama object, nama *object* dan *class* serta nama *class*.



- ***Found message*** merupakan suatu pesan yang men-stimulus terjadinya suatu skenario, asal dari found message ini sebenarnya tidak terlalu dipermasalahkan, karena diagram ini tidak menekankan pada **apa yang menginisiasi** suatu skenario (hal ini ditekankan dalam diagram use case), tetapi pada **skenario apa yang terjadi** bila sudah di inisiasi. Found message bisa berasal dari main program atau aktifitas user.
- ***Activation bar*** atau focus of control dalam setiap life line menunjukkan kapan suatu instance aktif dalam interaksi, activation bar ini juga berhubungan dengan fungsi dari instance yang berada dalam stack.
- ***Lifeline*****:** merupakan jalur hidup suatu instance kelas tertentu, selain berfungsi untuk mengetahui kapan suatu instance hidup dan dihapus, juga untuk melinierkan urutan pemanggilan pesan untuk instance yang bersangkutan. *Lifeline* digambarkan sebagai garis vertikal. *Lifeline* merepresentasikan eksistensi sebuah peran (*role*) pada waktu tertentu. Simbole peran digambarkan pada bagian atas dari *lifeline,* dan menunjukkan nama dan tipenya, dipisahkan dengan sebuah tanda titik dua (*colon*).
- ***Asynchronous & synchronous message*** 

& **:** *message* adalah komunikasi antar peran (*roles*). *Message* digambarkan sebagai garis panah *solid* antara *lifelines* dari dua peran. Panah *synchronous message* dimulai dan berakhir pada *lifeline* yang sama, *synchronous message* merupakan pesan yang dikirim dari satu lifeline ke lifeline lainnya, dimana si pengirim pesan akan menunggu sampai adanya balasan dari penerima (ada pesan balasan), sedangkan *asynchronous message* berakhir pada *lifeline* yang berbeda; *asynchronous message* merupakan pesan yang dikirim antar dua lifeline dimana pengirim tidak perlu menunggu adanya pesan balasan.. Panah dilabelkan dengan nama dari *message* dan parameternya. Panah bisa juga dilabelkan dengan nomor urut.
- ***Interaction Use*** 

adalah referensi/acuan untuk sebuah interaksi yang ada di dalam definisi dari interaksi lain. Kotak persegi panjang menutupi *lifeline* yang termasuk di dalam interaksi yang diacu.
- ***Alternative Combined Fragment*** 

: memodelkan aksi-aksi yang mungkin dilakukan yang akan dieksekusi jika *guard condition* terpenuhi.
- ***Option Combined Fragment*** 

: digunakan untuk memodelkan urutan kejadian yang masuk pada suatu kondisi tertentu, yang akan terjadi; atau sebaliknya, urutan kejadian tersebut tidak terjadi. ***Option*** pada dasarnya untuk menggambarkan *statement* "**if-then"**

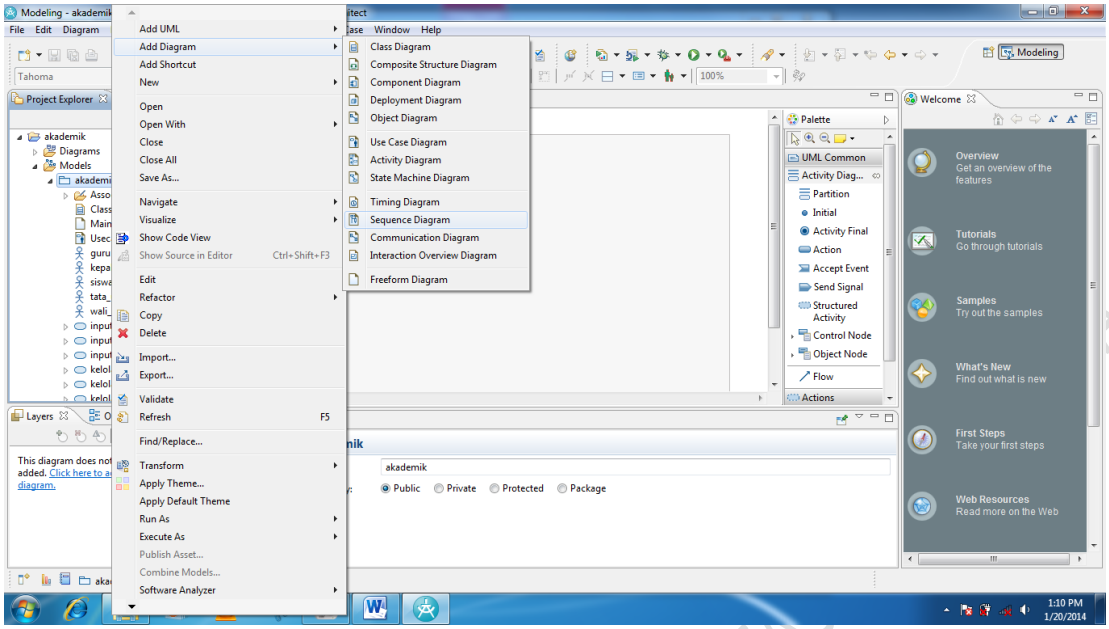
- ***Loop Combination Fragment*** 

: elemen ini sangat mirip dengan *option combined fragment. Combination fragment* ini digunakan untuk memodelkan urutan kejadian yang berulang (*repeating sequence*).
- ***Object*** didapat dari class *diagram*
- ***Found message*** merupakan suatu pesan yang men-stimulus terjadinya suatu skenario, asal dari found message ini sebenarnya tidak terlalu dipermasalahkan, karena *diagram* ini tidak menekankan pada **apa yang menginisiasi** suatu skenario (hal ini ditekankan dalam *diagram use case*), tetapi pada **skenario apa yang terjadi** bila sudah di inisiasi. Found message bisa berasal dari main program atau aktifitas user.
- ***Activation bar*** atau *focus of control* dalam setiap life line menunjukkan kapan suatu instance aktif dalam interaksi, activation bar ini juga berhubungan dengan fungsi dari instance yang berada dalam stack.
- ***Self call*** identik dengan procedure call, hanya saja dipanggil dalam fungsi milik instance sendiri yang sedang aktif pada saat itu (ditandai dengan activation bar).
- ***Return value*** merupakan suatu nilai balik yang dihasilkan dari pemanggilan suatu fungsi, return value tidak selalu harus digambar, gambarkan hanya bila menambah informasi tambahan untuk memahami skenario yang terjadi.
- ***Responsibility*** merupakan letak perbedaan diantara dua diagram tersebut, pada diagram pertama, terlihat bahwa sebagian besar pemanggilan fungsi dilakukan oleh instance FormPinjam (over responsibility), ini yang disebut dengan centralized control. Pada diagram kedua beberapa pemanggilan *disebar* ke instance yang lain, sehingga masing-masing instance melakukan proses **yang seharusnya** mereka lakukan, ini yang disebut dengan distributed control. Responsibility dari suatuclass ditentukan oleh konteksclass tersebut dalam suatu skenario, diagram kedua lebih membagi responsibility ke instance-instance yang terlibat, dan control seperti ini lebih disukai dalam pendekatan object oriented, karena pada proses design kita memang seharusnya bisa membagi responsibility dari kelas-kelas yang kita rancang.

4.1.2 LANGKAH – LANGKAH MEMBUAT SEQUENCE DIAGRAM

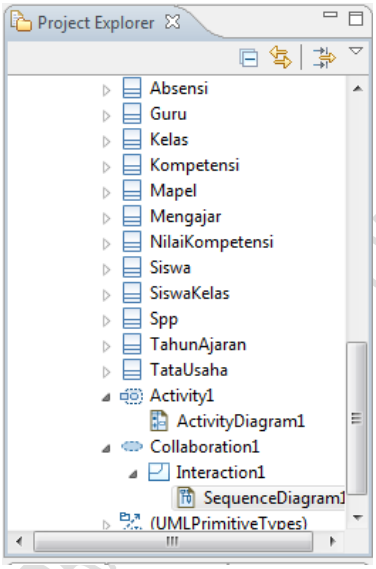
Berikut merupakan langkah-langkah dalam membuat sequence diagram :

Klik kanan pada file yang telah dibuat (misal: akademik) > Add Diagram > klik Sequence Diagram

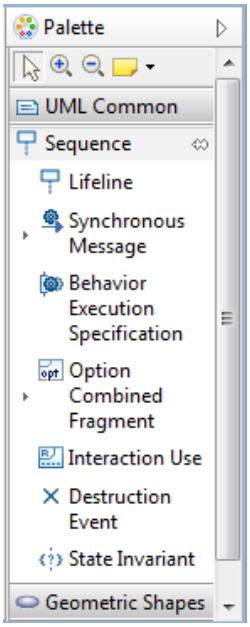


Gambar 4-2 Menu pilihan add sequence diagram

Setelah itu akan tampil halaman seperti di bawah ini

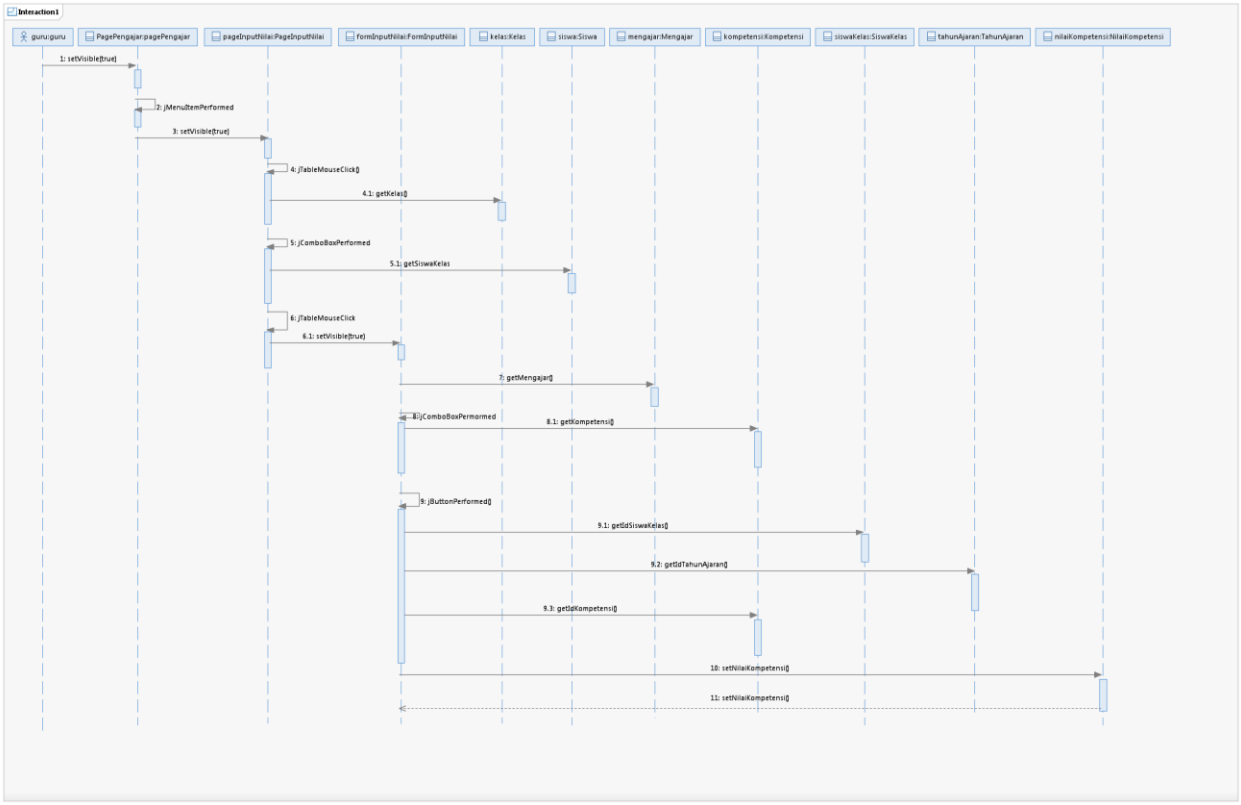


Gambar 4-3 Project explorer sequence diagram

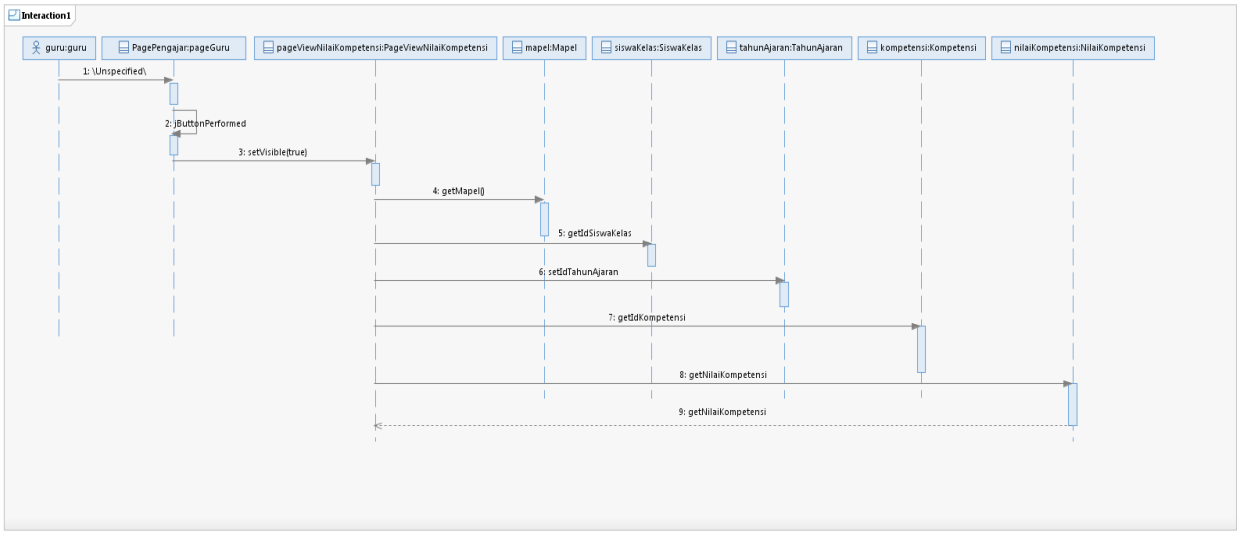


Gambar 4-4 atribut sequence diagram di RSA

Berikut contoh *Sequence Diagram*: Peminjaman untuk studi kasus akademik



Gambar 4-5 sequence diagram input penilaian



Gambar 4-6 sequence diagram view penilaian

Diagram sequence secara explicit menggambarkan interaksi antar object yang terlibat dalam suatu skenario, berdasarkan urutan waktu. Terlihat, instansiasi dari masing-masingclass yang terlibat saling mengirimkan dan menerima pesan, pesan yang dikirimkan berupa pemanggilan prosedur atau pengembalian suatu nilai balik/status.

4.2 COMMUNICATION DIAGRAM

Sebuah Diagram Komunikasi menggambarkan pola interaksi antara peran dalam sebuah kolaborasi. Ini menunjukkan partisipasi mereka dalam interaksi dengan hubungan mereka satu sama lain, dan oleh pesan-pesan yang mereka kirimkan satu sama lain.

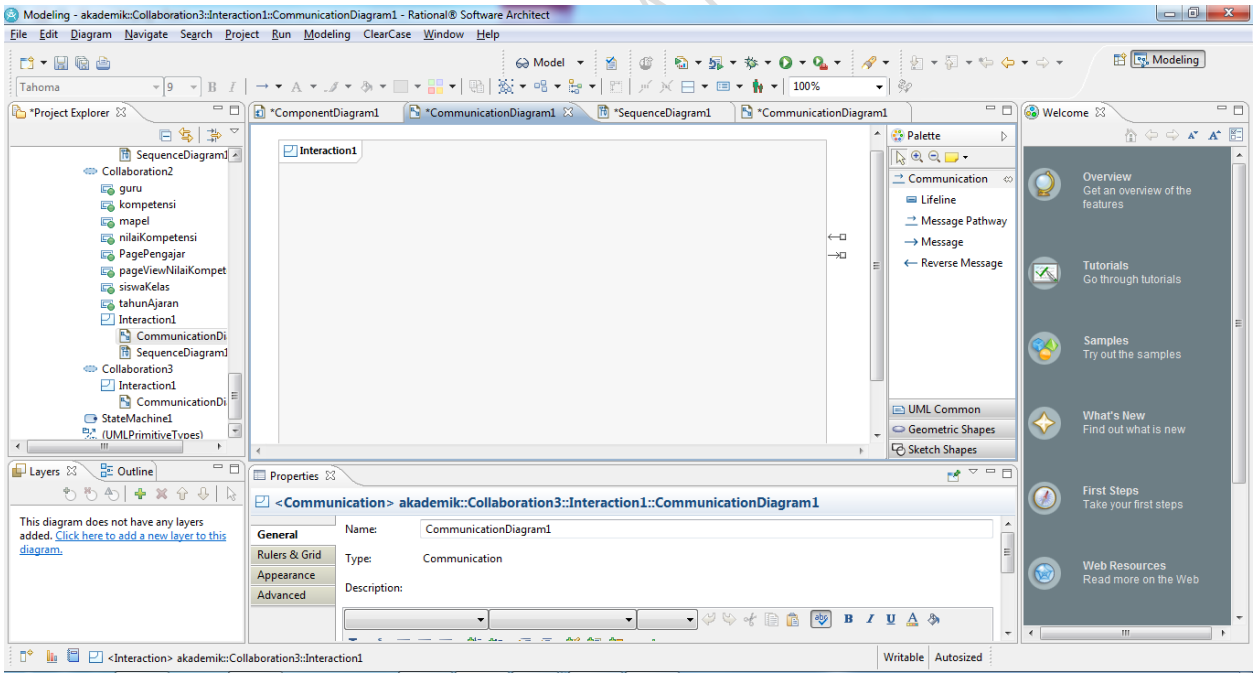
4.2.1 KOMPONEN-KOMPONEN PADA COMMUNICATION DIAGRAM

1. ***Lifeline*** mewakili peran yang berkomunikasi satu sama lain dalam diagram komunikasi. *Lifeline* dapat berupa ***class,* aktor, komponen, atau peran yang dapat ditentukan.**
2. ***Message*** adalah komunikasi antara peran yang terdapat dalam diagram komunikasi. *Message* pada diagram komunikasi terbagi atas 2 :
    a. ***Synchronous Message*** adalah pesan yang disampaikan satu *lifeline* dimana pesan ini membutuhkan jawaban dari *lifeline* yang menerima
    b. ***Asynchronous Message*** adalah pesan yang disampaikan tanpa membutuhkan jawabah dari *lifeline* penerima.

4.2.2 LANGKAH – LANGKAH MEMBUAT COMMUNICATION DIAGRAM

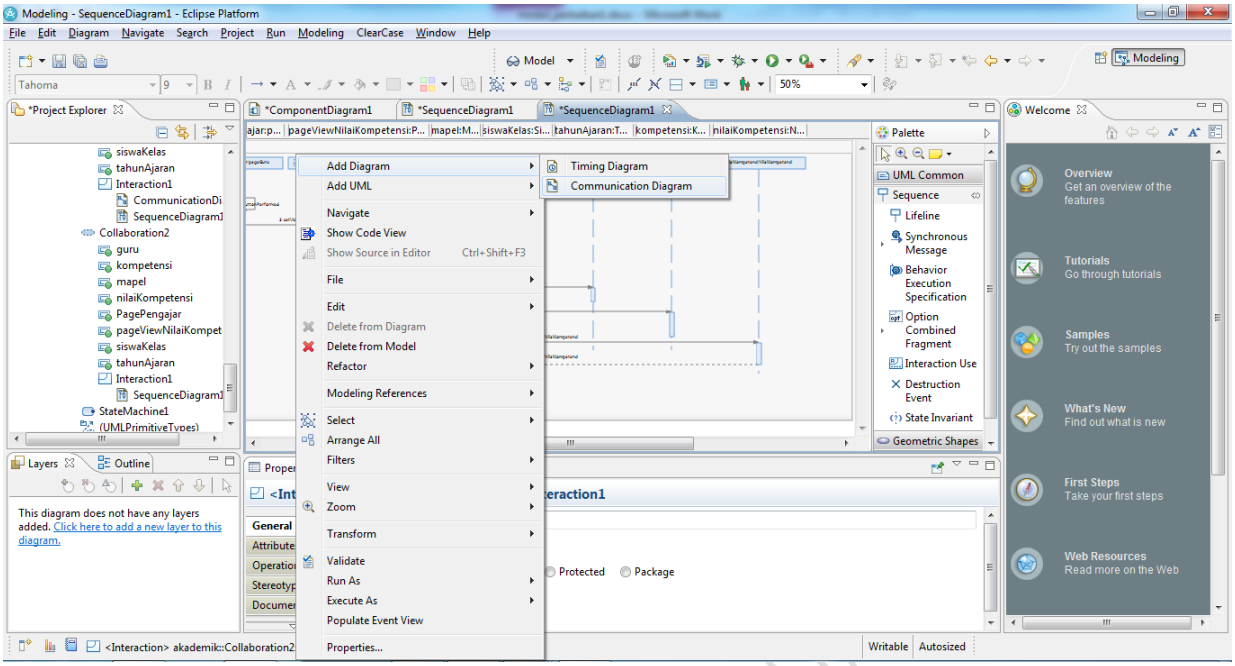
Cara 1 : Klik kanan pada file yang telah dibuat (misal: akademik) > Add Diagram > klik Communication Diagram

Setelah itu akan tampil halaman seperti di bawah ini



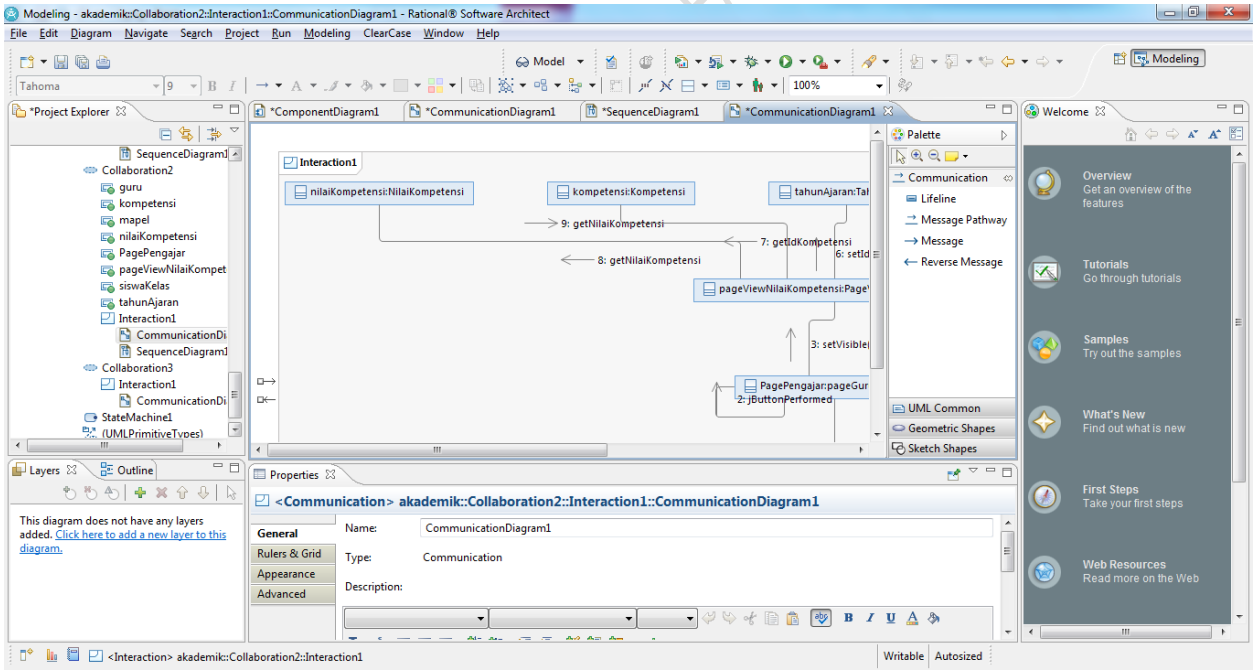
Gambar 4-7 gambar communication diagram <u>tanpa</u> dibentuk dari sequence diagram

Cara 2 : Klik kanan pada sequence diagram yang telah dibuat > Add Diagram > klik Communication Diagram



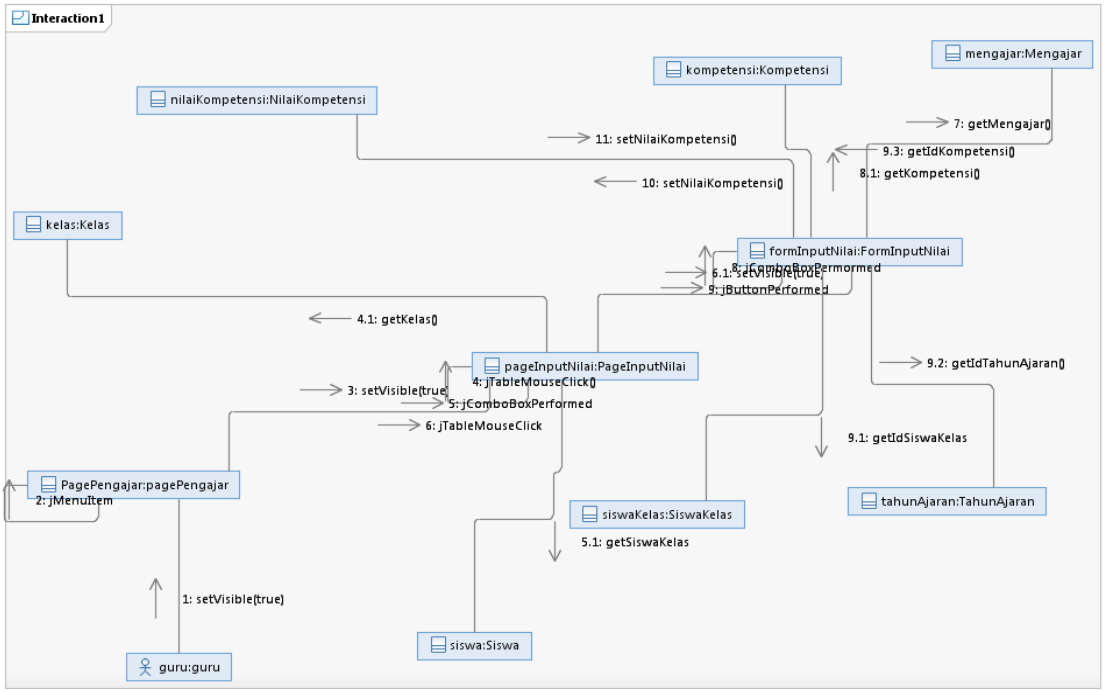
Gambar 4-8 gambar add diagram communication diagram

Setelah itu akan tampil halaman seperti di bawah ini

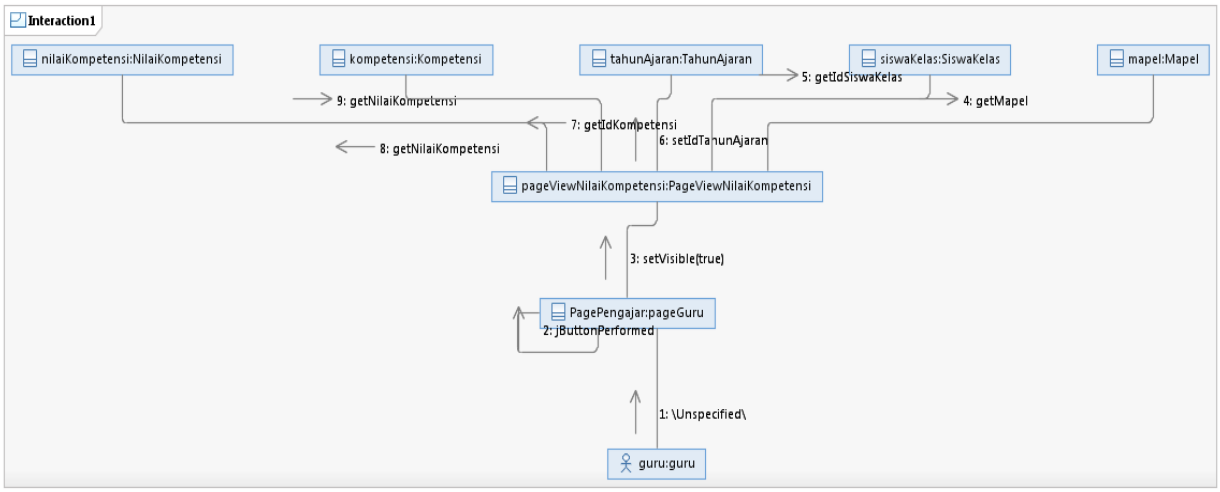


Gambar 4-9 gambar communication diagram yang dibentuk dari sequence diagram

Berikut contoh Communication Diagram :



Gambar 4-10 input penilaian



Gambar 4-11 view penilaian

MODUL 5 ACTIVITY DIAGRAM

Tujuan Praktikum::

1. Praktikan dapat memahami konsep Acitivity Diagram
2. Praktikan dapat menggambarkan *activity diagram* dan membedakannya dengan *state diagram*

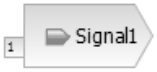
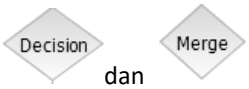
5.1 ACTIVITY DIAGRAM

Activity diagram adalah jenis khusus dari *Statechart diagram*, menunjukkan flow aktifitas ke aktifitas (bukan status ke status). Oleh karena itu biasakan menggunakan kata kerja aktif dalam penamaan suatu aktifitas dan bedakan dengan state diagram yang biasanya menyatakan suatu kondisi atau keadaan (kata sifat atau keterangan).

Activity diagram memodelkan *workflow* proses bisnis dan urutan aktifitas dalam sebuah proses. Diagram ini sangat mirip dengan *flowchart* karena memodelkan *workflow* dari satu aktifitas ke aktifitas lainnya atau dari aktifitas ke status. Menguntungkan untuk membuat *activity diagram* pada awal pemodelan proses untuk membantu memahami keseluruhan proses. *Activity diagram* juga bermanfaat untuk menggambarkan *paralel behavior* atau menggambarkan interaksi antara beberapa *use case*.

5.1.1 ELEMEN - ELEMEN ACTIVITY DIAGRAM

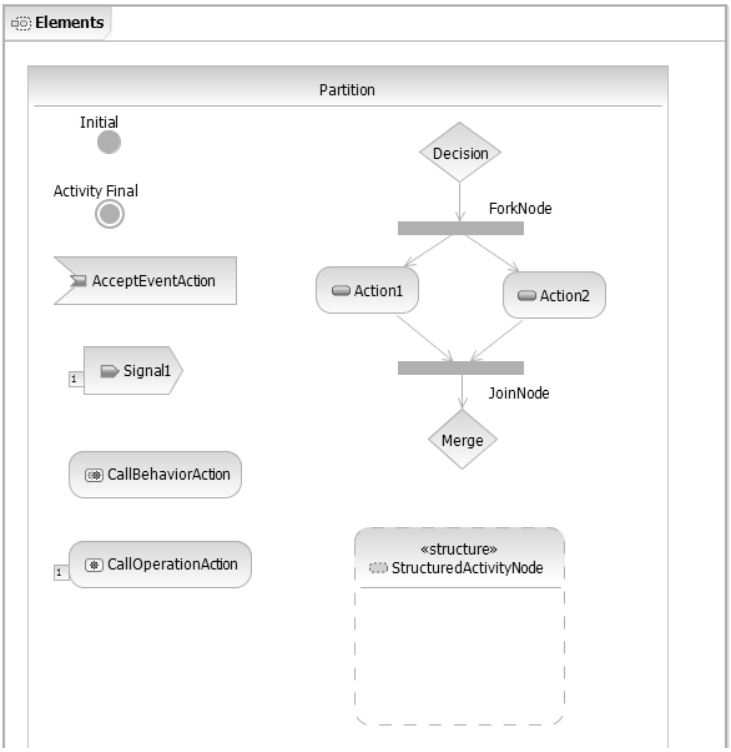
Elemen-elemen *activity diagram* pada RSA:

No	Nama Elemen	Gambar Elemen	Keterangan Elemen
1	*Initial* (mulai)		Meruipakan node awal dari proses Activity Diagram.
2	*Activity final* (akhir)		Merupakan node akhir dari proses Activity Diagram. Status *end* boleh lebih dari 1.
3	*Action node*		Menunjukkan proses atomik atau transformasi yang dilakukan oleh sistem yang sedang dimodelkan.
4	*Accept Event Action*		Adalah aksi-aksi yang membuat dan mengirimkan sinyal untuk objek target.
5	*Send Signal Action*		Adalah aksi-aksi yang membuat dan mengirimkan sinyal untuk objek target.
6	*Decision* dan *Merge*	 dan	*Decision* menyediakan percabangan untuk melakukan operasi yang dibutuhkan berdasarkan pada input atau *system state. Merge* mengembalikan *flow* yang bercabang menjadi satu *flow* bersama.

No	Nama Elemen	Gambar Elemen	Keterangan Elemen
7	*Fork* dan *Join*		Percabangan juga ditunjukkan sebagai *flow* yangmelewati *node Fork* dan digabungkan oleh *node Join. Nodes* ini biasanya digunakan setelah *node Decision*
8	*Call Behavior Action*	CallBehaviorAction	Merepresentasikan panggilan yang terhubung dengan *Activity* yang terpisah dan *diagram*nya. Dengan dua kali meng-klik *Call Behavior Action* akan membuka *diagram* yang berhubungan dengan *action* tersebut pada *editor*.
9	*Call Operation Action*	1 CallOperationAction	Meminta satu operasi dari satu *Class* yang spesifik (*interface, subsystem,* dll.). Nama *action* tersebut harus nama operasi yang sedang dipanggil.
10	*Structured Activity Node*	«structure» StructuredActivityNode	Unit dasar pada satu aktivitas adalah *node Action. Node Action* menunjukkan proses atomik atau transformasi yang dilakukan oleh sistem yang sedang dimodelkan.

Tabel 5-1 Elemen-elemen activity diagram pada RSA

Untuk lebih jelasnya, bisa dilihat pada gambar berikut :

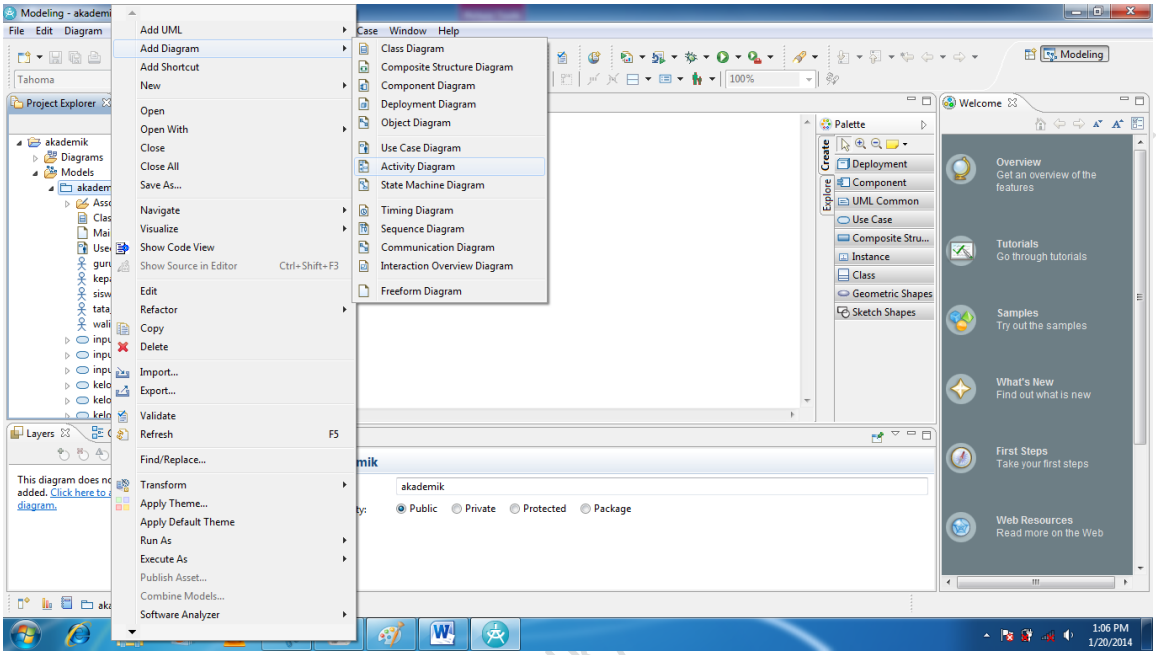


Gambar 5-1 Elemen-elemen *Activity Diagram* pada RSA

5.2 LANGKAH-LANGKAH MEMBUAT ACTIVITY DIAGRAM

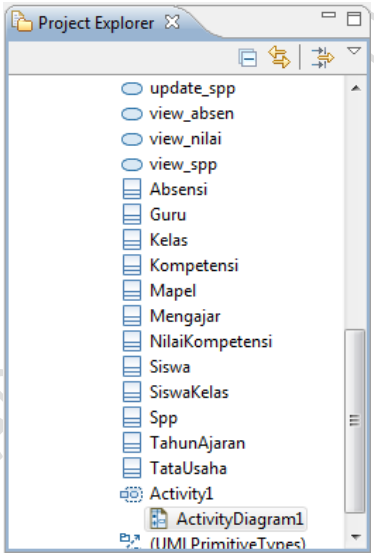
Berikut merupakan langkah-langkah dalam membuat class diagram :

Klik kanan pada sequence diagram yang telah dibuat > Add Diagram > klik Activity Diagram

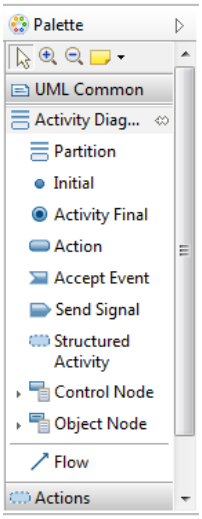


Gambar 5-2 Pilihan add diagram pada activity diagram

Setelah itu akan tampil halaman seperti di bawah ini



Gambar 5-3 Project Explorer activity diagram



Gambar 5-4 Komponen activity diagram di RSA

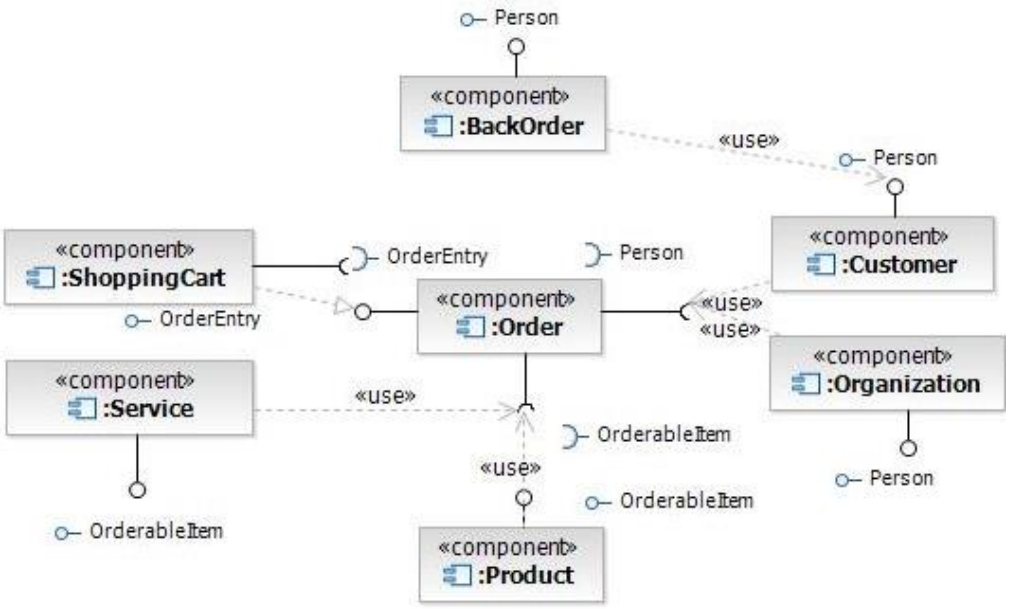
MODUL 6 COMPONENT DIAGRAM & DEPLOYMENT DIAGRAM

Tujuan Praktikum::

Mampu membuat Component dan Deployment Diagram

6.1 COMPONENT DIAGRAM

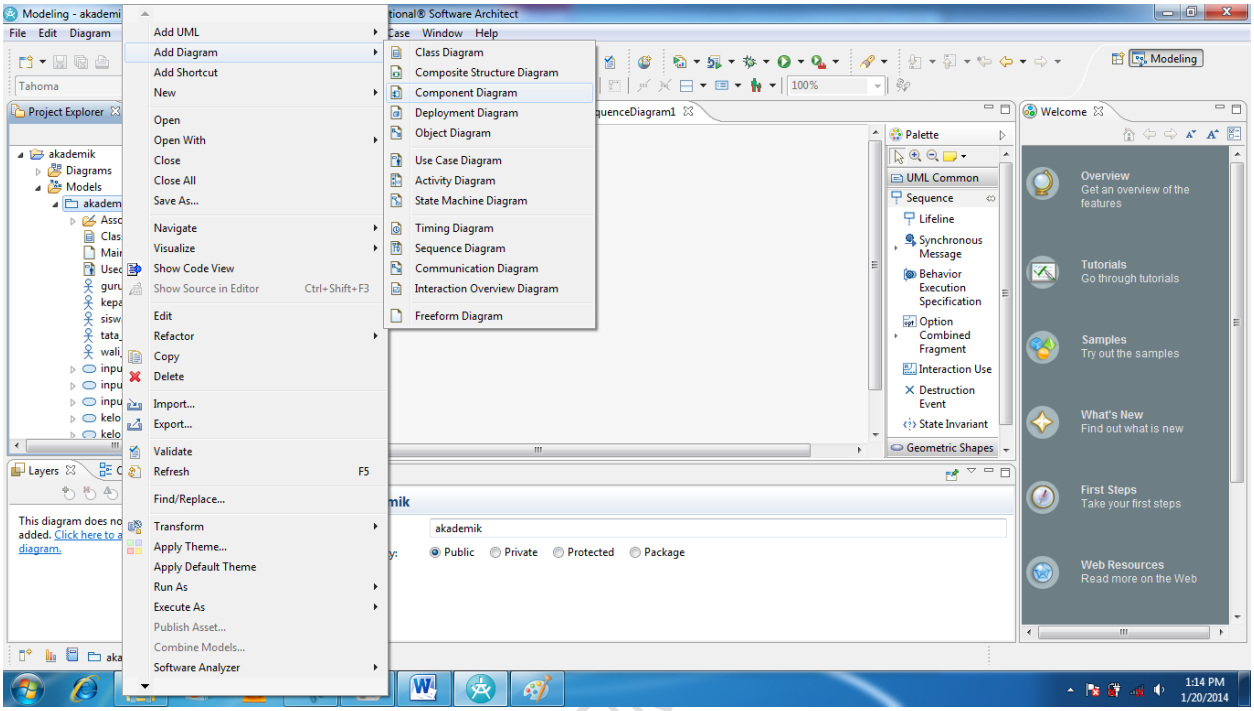
Component diagram menggambarkan struktur yang mewakili unsur independen, bagian dipertukarkan dari suatu sistem. Mereka sesuai dan menyadari satu atau lebih interface disediakan dan diperlukan, yang menentukan perilaku komponen.Komponen kasus pada pemodelan UML, contoh komponen adalah model elemen yang mewakili entitas aktual dalam suatu sistem. Pada compenent diagram mewakili paket, artefak dan antarmuka suatu sistem



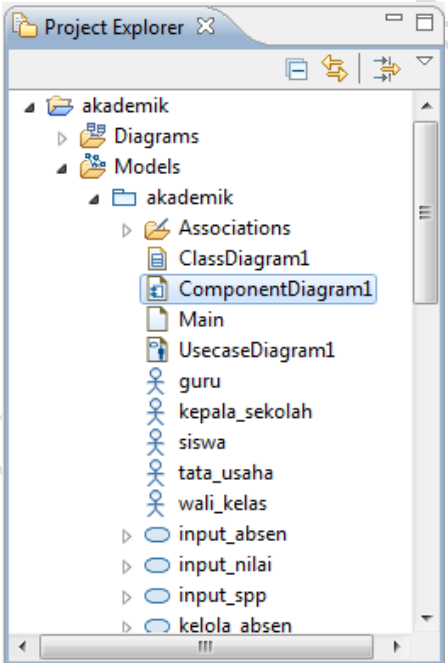
Gambar 6-1 Contoh Component Diagram belanja online

6.2 LANGKAH-LANGKAH MEMBUAT COMPONENT DIAGRAM

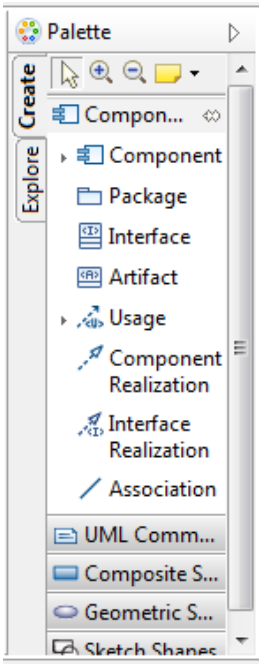
Klik kanan pada file yang telah dibuat (misal: akademik) > Add Diagram > klik Component Diagram



Setelah itu akan tampil halaman seperti di bawah ini



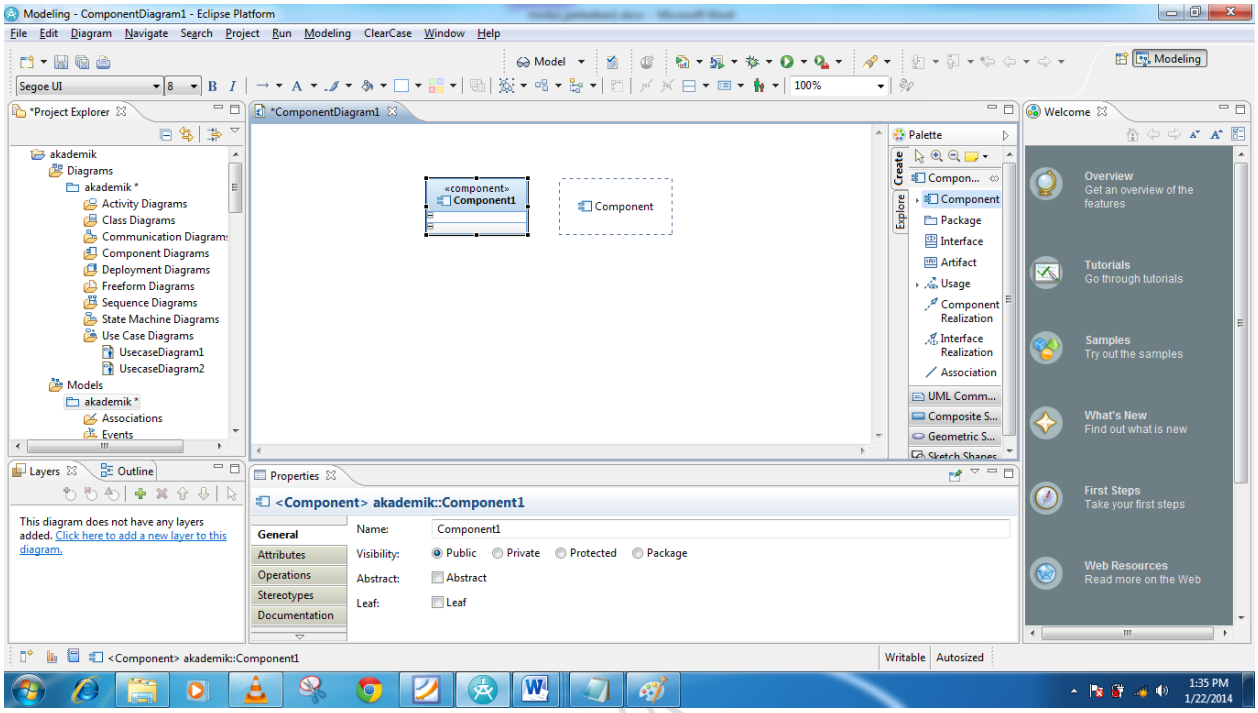
Gambar 6-2 Project explorer component diagram



Gambar 6-3 Komponen pada component diagam di RSA

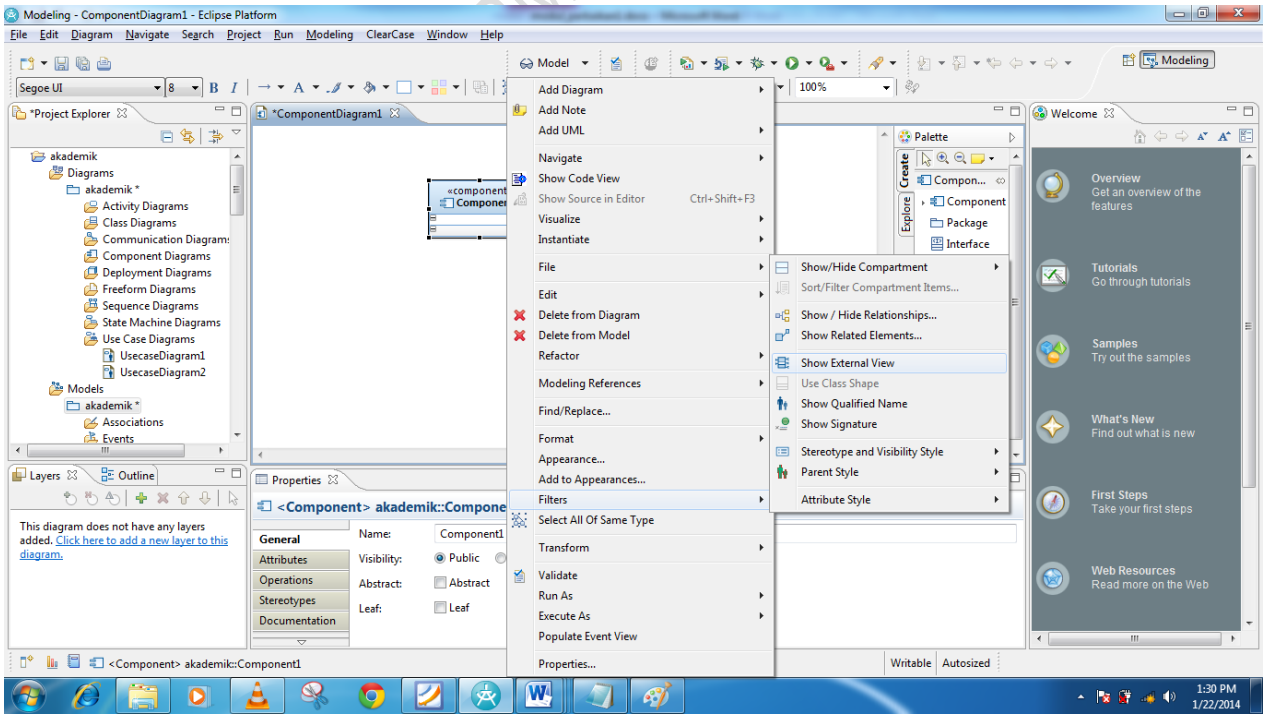
Setelah terbikin blank baru maka:

1. klik main page > kemudian drag icon component diagram



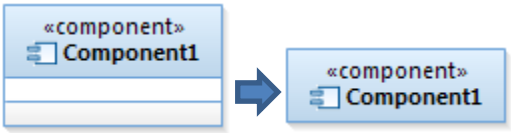
Gambar 6-4 Halaman pengerjaan component diagram

2. Setelah component diagram terbuat > klik component tadi dan rubah jadi external view. Dengan melakukan klik kanan > filters > show external view

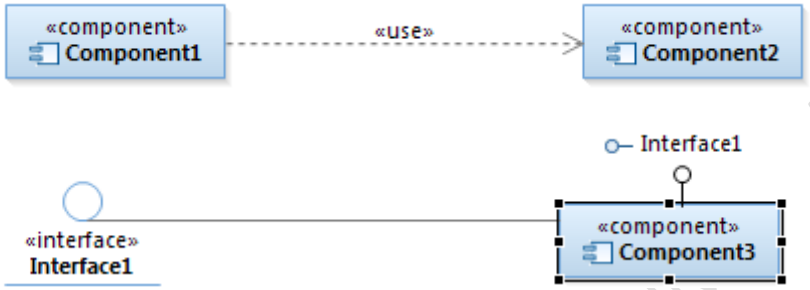


Gambar 6-5 Show external view pada component diagram

Maka akan tampil seperti di bawah ini



Gambar 6-6 tampilan show external view



Gambar 6-7 tampilan show external view dan use class shape

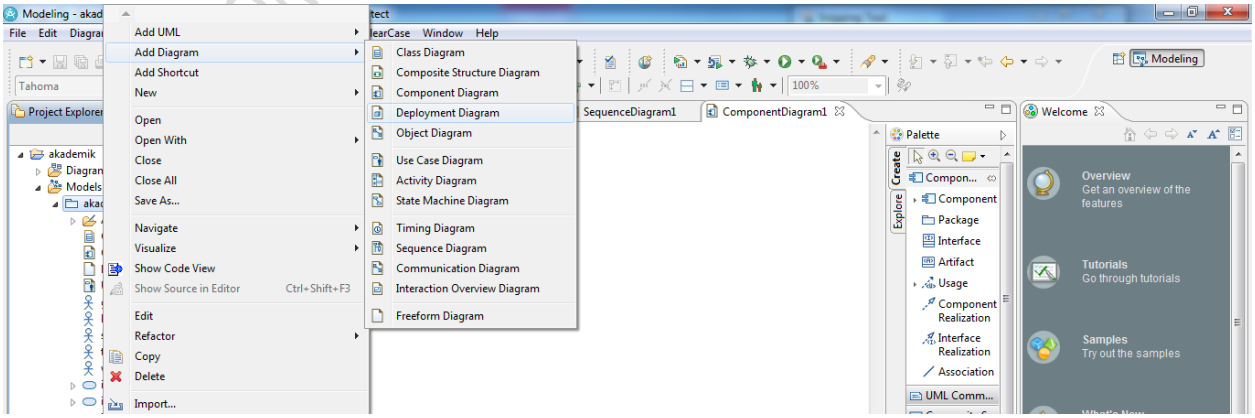
6.3 DEPLOYMENT DIAGRAM

Deployment/physical diagram menggambarkan detail bagaimana komponen di-*deploy* dalam infrastruktur sistem, di mana komponen akan terletak (pada mesin, server atau piranti keras apa), bagaimana kemampuan jaringan pada lokasi tersebut, spesifikasi server, dan hal-hal lain yang bersifat fisikal Sebuah *node* adalah server, *workstation*, atau piranti keras lain yang digunakan untuk men-*deploy* komponen dalam lingkungan sebenarnya. Hubungan antar *node* (misalnya TCP/IP) dan *requirement* dapat juga didefinisikan dalam diagram ini.

Dalam menentukan apakah *node* tersebut merupakan *processor* atau *device* adalah berdasarkan fungsionalitasnya. Untuk processor adalah perangkat keras yang digunakan sebagai pengolahan objek-objek pada system, sedangkan device adalah perangkan keras yang akan menghubungkan processor (misalnya network) serta mendukung tugas dari processor (misalnya printer).

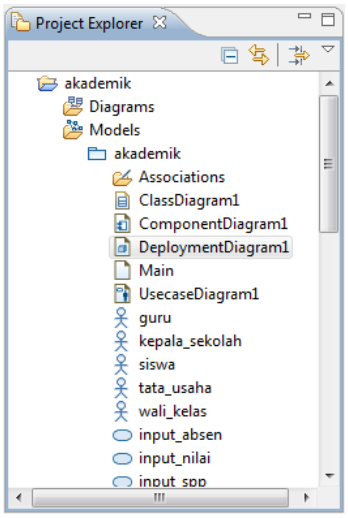
6.4 LANGKAH-LANGKAH MEMBUAT DEPLOYMENT DIAGRAM

Klik kanan pada file yang telah dibuat (misal: akademik) > Add Diagram > klik Deployment Diagram

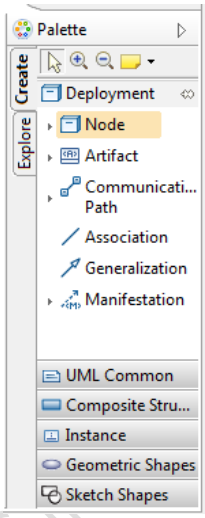


Gambar 6-8 Menu pilihan deployment diagram

Setelah itu akan tampil halaman seperti di bawah ini



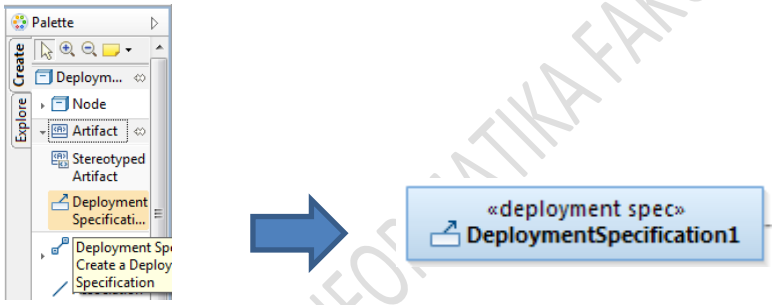
Gambar 6-9 Project explorer dep



Gambar 6-10 Komponen deployment diagram

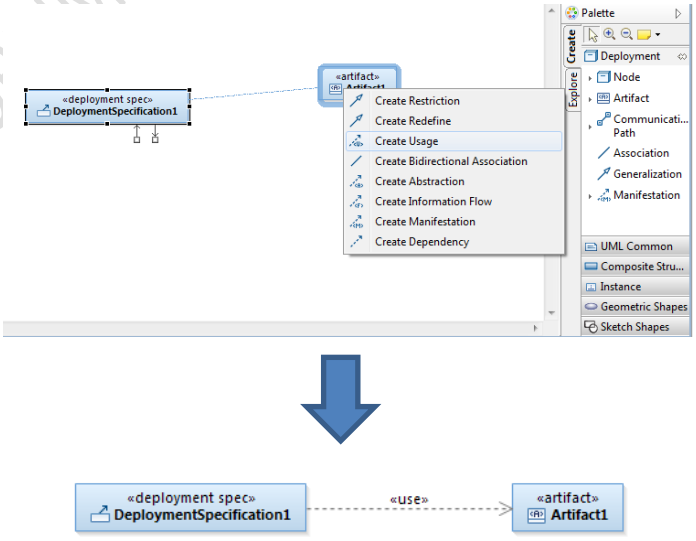
Menambahkan Connection

1. Pada pallete Klik *Artifact* > Deployment Specificstion



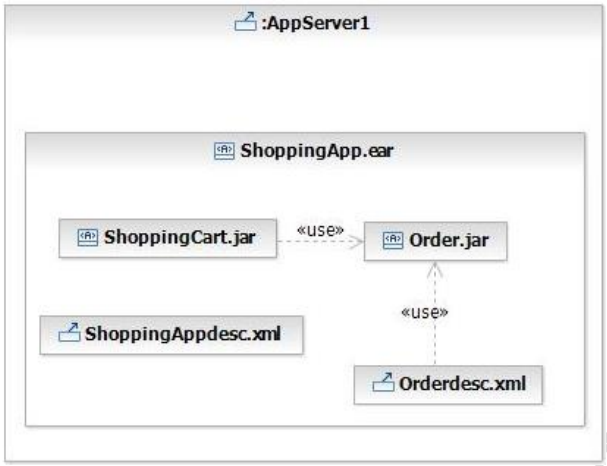
Gambar 6-11 Pilih deployment specification

2. Kemudian klik artifact. Klik **Create Usage**



Gambar 6-12 Create usage

Contoh deployment diagram :



Gambar 6-13 contoh deployment diagram akademik

MODUL 7 STATE DIAGRAM

Tujuan Praktikum :

Mahasiswa mampu membuat State Diagram

7.1 STATE MACHINE DIAGRAM

- *State Transition Diagram* adalah diagram yang digunakan untuk menggambarkan *behavior* atau siklus hidup dari suatu *class*, dengan menunjukkan adanya perubahan *state* di suatu *class* berdasarkan *event* dan *message* yang dikirimkan dan diterima oleh *class* tersebut.
- *Behavior* yang menggambarkan urutan *state* dari *object* sepanjang waktu hidup-nya; *event* apa saja yang tejadi dan seperti apa respon *object* terhadap *event* tersebut, serta bagaimana transisi antara *state*-nya.
- Menceritakan apa yang bisa suatu *object* rasakan, lihat, dan lakukan selama *object* tersebut hidup.

State diagram tidak perlu dibuat untuk setiap *class* di sistem. *State diagram* hanya perlu dibuat untuk *class* atau *object* **yang berkelakuan dinamis**. *Object* yang dinamis adalah *object* yang memiliki sekumpulan *response* yang berbeda terhadap satu atau beberapa *event* (*behavior* dari *object* tsb selalu berubah). Kita dapat melihat apakah suatu *object* bersifat dinamis atau tidak dengan cara mempelajari satu, atau beberapa *interaction diagram* berbeda yang melibatkan object tersebut. *Object* yang dinamis biasanya ditandai dengan *object* yang mengirim dan menerima beberapa pesan. Selain itu objek yang dinamis juga melibatkan objek yang memiliki kondisi lebih dari satu.

State diagram menunjukkan :

- *state-state* dari *object* tunggal
- *event-event* atau pesan yang menyebabkan transisi dari satu *state* ke *state* yang lain
- *action* yang merupakan hasil dari perubahan sebuah *state*

7.2 KOMPONEN STATE MACHINE DIAGRAM

Semua komponen yang dibutuhkan dalam pembuatan state machine diagram berada pada pallete:

7.2.1 STATES

Digambarkan berbentuk segi empat dengan sudut membulat dan memiliki nama sesuai kondisinya saat itu

Gambar 7-1 *Notasi UML untuk state*

Membuat state:

1. Klik untuk memilih *icon state* dari *pallete*.
2. Klik untuk menempatkan *state* pada *state transition diagram*.
3. Dengan *state* masih dipilih, masukkan nama *state*.

7.2.2 TRANSITION

Sebuah *transition* direpresentasikan oleh sebuah panah yang menunjuk dari *state* awal ke *state* berikutnya (*state* yang dituju).

Gambar 7-2 *transition*

Membuat Transition:

1. Klik untuk memilih *icon transition* dari *pallete*.
2. Klik pada *state* asaldi *state machine diagram*.
3. *Drag transition* menuju *state* yang diinginkan (*state* tujuan).
4. Jika *transition* merupakan transisi yang mempunyai nama, masukkan nama ketika panah *state transition* masih dipilih.

Transition Details

Sebuah *transition* dapat mempunyai (*optional*) sebuah *action* dan/atau sebuah kondisi penjaga (*guard condition*) yang terasosiasi dengannya, dan mungkin juga memunculkan sebuah *event*. Sebuah *action* adalah kelakuan yang terjadi ketika *transition* terjadi. Sebuah *event* adalah pesan yang dikirim ke *object* lain di sistem. Kondisi penjaga adalah ekspresi *boolean* dari nilai atribut-atribut yang mengijinkan sebuah *transition* hanya jika kondisinya benar.

Gambar 7-3 transition hanya jika kondisinya benar

Untuk membuat detail transition, klik tanda panah. Tuliskan detail transitionnya, jika ada keterangan event, dapat disembunyikan.

7.2.3 SPECIAL STATES

Merupakan state untuk menggambarkan awal maupun akhir dari kejadian dalam suatu diagram statechart

Gambar 7-4 *Notasi UML untuk start dan stop state*

Membuat Start State

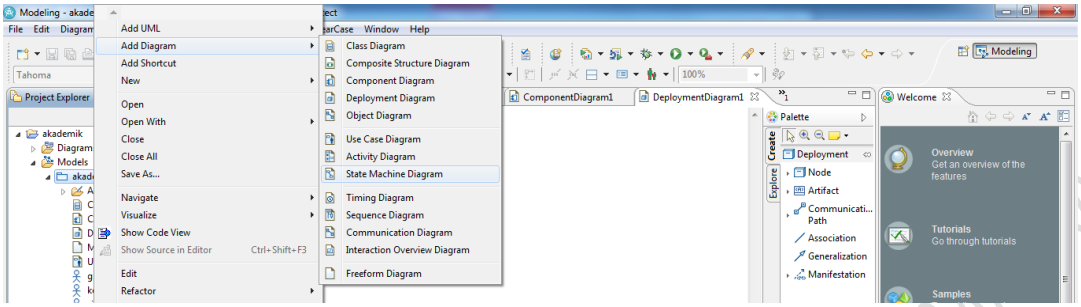
1. Klik untuk memilih *icon start state* dari pallete.
2. Klik pada *state machine diagram* untuk menggambarkan *icon start state*.
3. Klik untuk memilih *icon transition* dari *pallete*.
4. Klik pada *icon start state* dan *drag* panahnya menuju *state* yang diinginkan.

Membuat Stop State

1. Klik untuk memilih *icon stop state* dari *pallete*.
2. Klik pada *state machine diagram* untuk menggambarkan *icon stop state*.
3. Klik untuk memilih *icon transition* dari *pallete*.
4. Klik pada *state* dan *drag* panahnya menuju *icon stop state*.

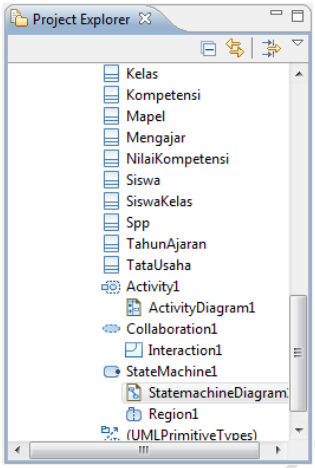
7.3 LANGKAH-LANGKAH MEMBUAT STATE MACHINE DIAGRAM

Klik kanan pada file yang telah dibuat (misal: akademik) > Add Diagram > klik State Machine Diagram

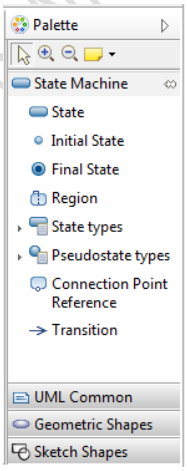


Gambar 7-5 Menu pilihan membuat state machine diagram

Setelah itu akan tampil halaman seperti di bawah ini

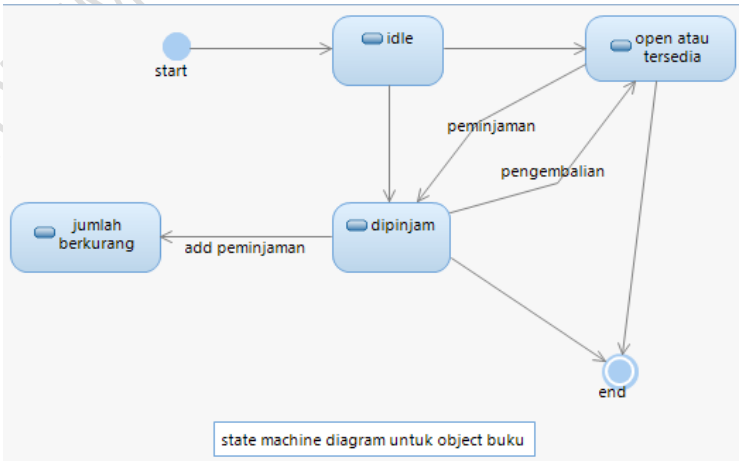


Gambar 7-6 Project explorer state machine diagram



Gambar 7-7 Komponen

Contoh state machine diagram :



Untuk self transition dapat dibuat dengan cara:

1. Klik icon transition dari pallete
2. Klik pada state yang hendak memiliki self transition

Modul 8 STRATEGY PATTERN

Tujuan Praktikum:

1. Mahasiswa memahami dasar-dasar *behavioral patterns*
2. Menggunakan *strategy pattern* pada suatu model
3. Menghubungkan elemen-elemen UML dengan parameter-parameter suatu *behavioral pattern*

8.1 PATTERNS

Pattern adalah gambaran solusi yang *reusable* dari persoalan umum yang muncul pada konteks tertentu. *Pattern* yang baik harus:

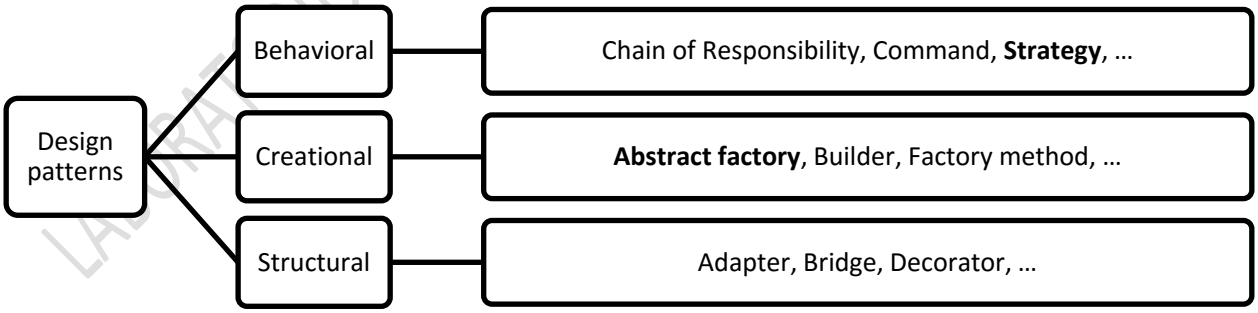
- bersifat umum
- mengandung solusi yang terbukti dapat menyelesaikan persoalan secara efektif pada konteks penerapannya
- dideskripsikan dalam bentuk yang mudah dipahami

Sebuah *design pattern* memiliki parameter-parameter untuk mengaitkan *pattern* ke model.

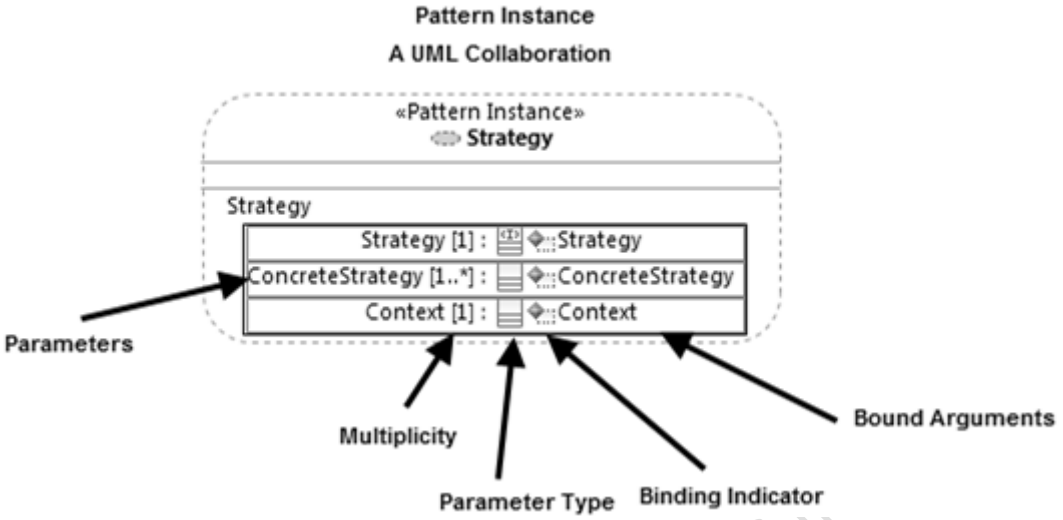
Design pattern dalam RSA dapat digunakan untuk menambahkan detail secara otomatis ke sebuah model, antara lain berupa (Help - Rational Software Architect, 2011):

- Elemen-elemen dan relasi model
  - *classes, packages,* dll
- Detail dari elemen-elemen model
  - *attributes, operations*, dll
- Markup dari model
  - *stereorypes* dari elemen-elemen model untuk membuat *analysis classes, subsystems,* dan *platform- and technology-specific elements.*

Design patterns secara umum terdiri atas 3 kelompok, yaitu: *behavioral, creational,* dan *structural*.



Notasi *design pattern* ditunjukkan pada Gambar berikut.



Gambar 8-1 Notasi *Design Pattern* pada Rational Software Architect

Pattern instances dalam RSA direpresentasikan dalam UML *collaborations* dengan stereotype <<pattern instance>>. Sebuah *pattern instance* memiliki fitur-fitur:

- *Parameters*: Tiap parameter memerlukan sebuah argumen. Jika terikat, icon berubah jadi kotak biru dengan panah ganda.
- *Parameter multiplicity*: ditunjukkan dalam tanda kurung [ ] setelah nama parameter.
- *Parameter type*: setelah multiplicity, sebuah icon atau teks menunjukkan tipe parameter (mis. class, interface, atau operation).
- *Binding indicator*: Sebuah icon atau teks yang menunjukkan apakah parameter telah memiliki argumen yang terkait. Kotak biru kosong menunjukkan bahwa belum ada argumen yang terkait pada parameter, sedangkan *binding icon* menunjukkan bahwa argumen telah terkait pada parameter.
- *Arguments*: satu (atau lebih, jika dimungkinkan dalam *pattern*) yang terikat pada parameter

8.2 STRATEGY PATTERN

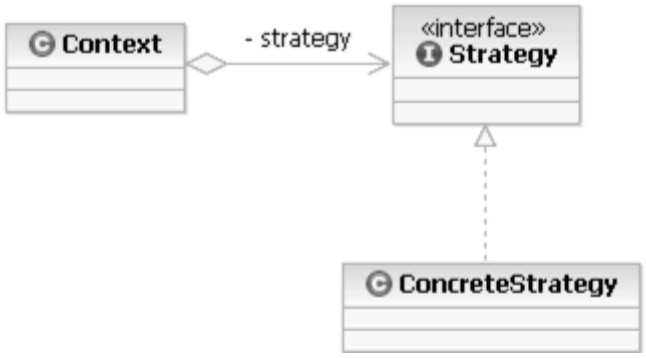
Strategy Pattern "mendefinisikan satu keluarga algoritma, mengenkapsulasi setiap algortima tersebut, dan membuat algoritma tersebut dapat dipertukarkan (interchangeable). Strategy pattern membedakan algoritma secara bebas dari klien-klien yang menggunakannya" (Freeman, Robson, & Bates, 2004). *Pattern* ini juga dikenal dengan nama *Policy*.

8.3 ELEMEN-ELEMEN STRATEGY PATTERN

- `Strategy` [1] → Mendeklarasikan sebuah interface umum untuk seluruh algoritma yang mendukung. Context menggunakan interface ini untuk memanggil algoritma yang didefinisikan olah ConcreteStrategy
- `ConcreteStrategy`: Class [1..*] → Mengimplementasikan algoritma dengan interface Strategy
- `Context` [1] → Dikonfigurasi dengan objek ConcreteStrategy. Memelihara satu referensi dengan objek Strategi. Bisa mendefinisikan sebuah interface yang mengizinkan Strategi mengakses datanya

8.4 OVERVIEW

Berikut gambaran overview singkat terkait strategy pattern

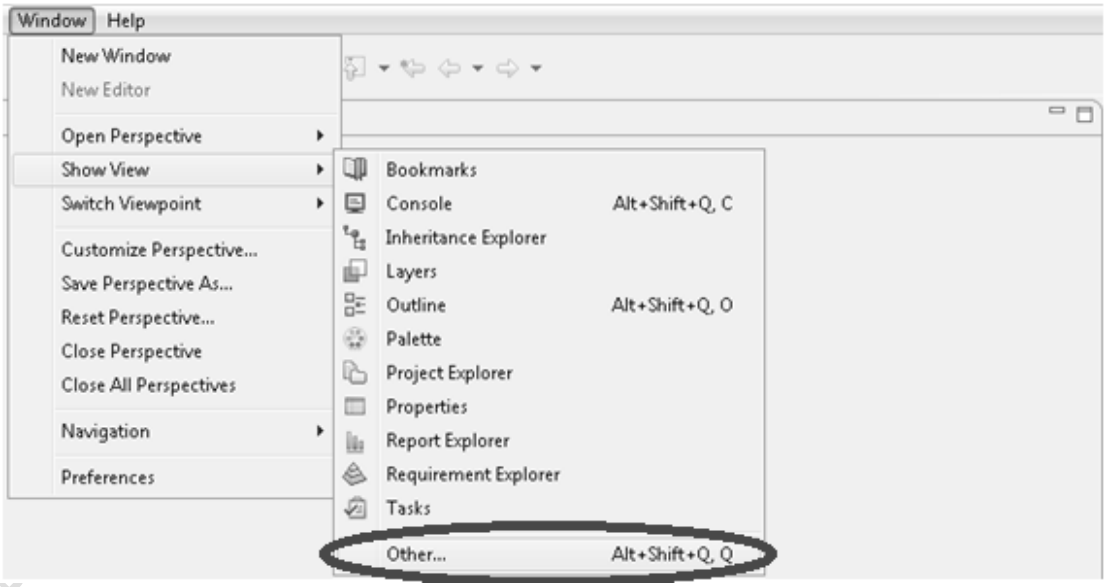


Gambar 8-2 Overview dari *Strategy Pattern* pada Rational Software Architect

8.4.1 PATTERN EXPLORER

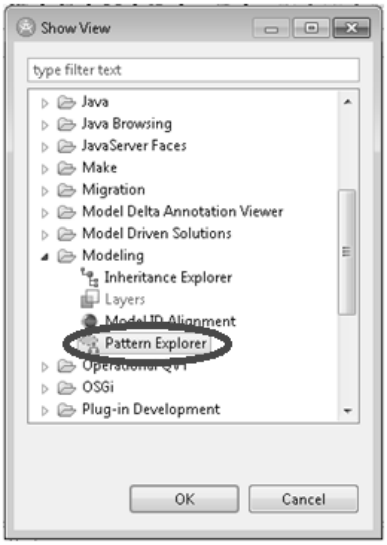
Strategy pattern dapat diterapkan dengan membuka *Pattern Explorer* pada Rational Software Architect (RSA). *Pattern Explorer* menampilkan daftar seluruh *patterns* yang tersedia, dan dapat digunakan untuk mengatur, memilih serta menerapkan *patterns.*

- Pilih *Window > Show View > Other...*



Gambar 8-3 Membuka *view Pattern Explorer*

- Pilih *Modeling > Pattern Explorer* pada *Show View* dan klik OK



Gambar 8-4 Memilih *view Pattern Explorer*

View Pattern Explorer dapat dipindah-pindahkan di posisi sesuai dengan yang kita inginkan.

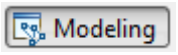
8.3 LANGKAH-LANGKAH MEMBUAT STRATEGY PATTERN

Berikut langkah-langkah dalam pembuatan strategy pattern :

8.3.1 MEMBUAT PROYEK UML MODEL

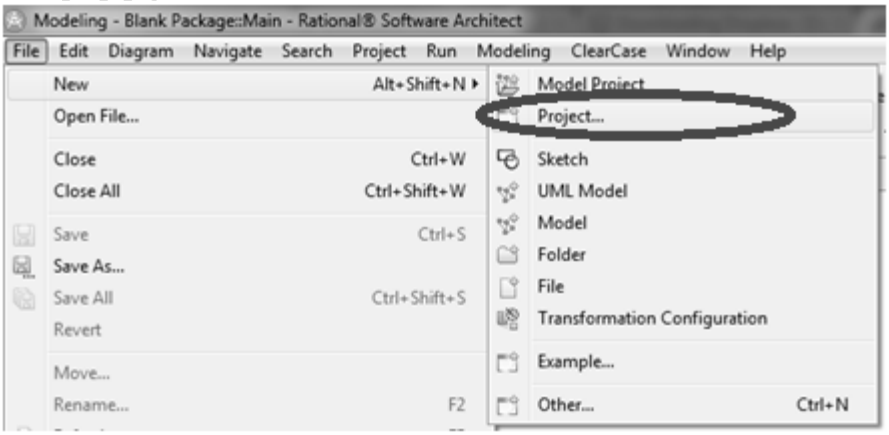
Sebelum menerapkan sebuah *pattern*, kita harus terlebih dahulu membuat proyek pemodelan UML. Berikut ini adalah langkah-langkahnya:

(1) Tutup seluruh model yang terbuka (jika ada).

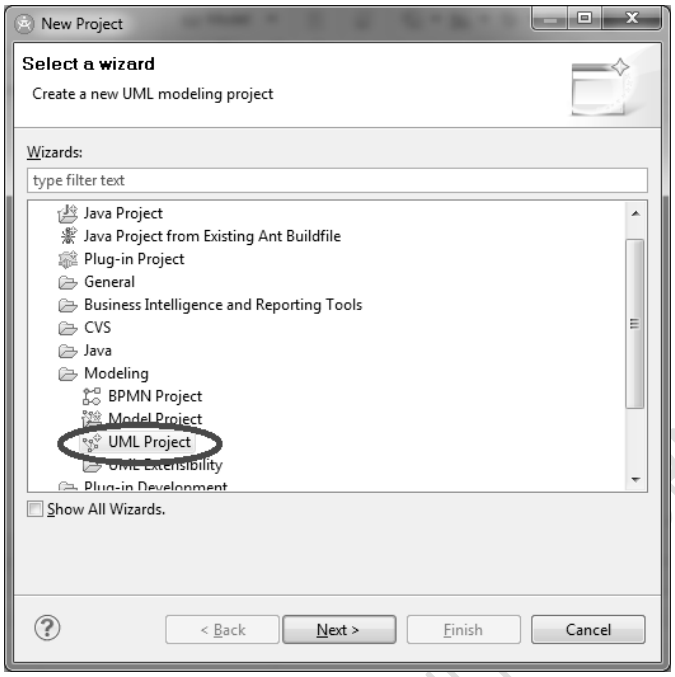
(2) Aktifkan perspektif *Modelling,* dengan meng-klik icon 

di sisi kanan atas layar.

(3) Pilih *File > New > Project..* di menu utama dan pilih "*UML Project*" pada dialog box "*New Project*". Lalu klik *Next*.

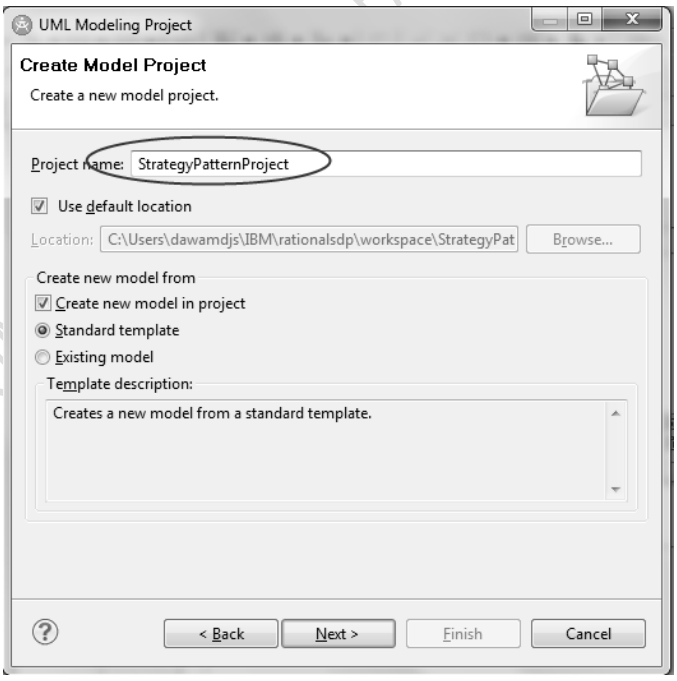


Gambar 8-5 Membuat *UML Project baru*



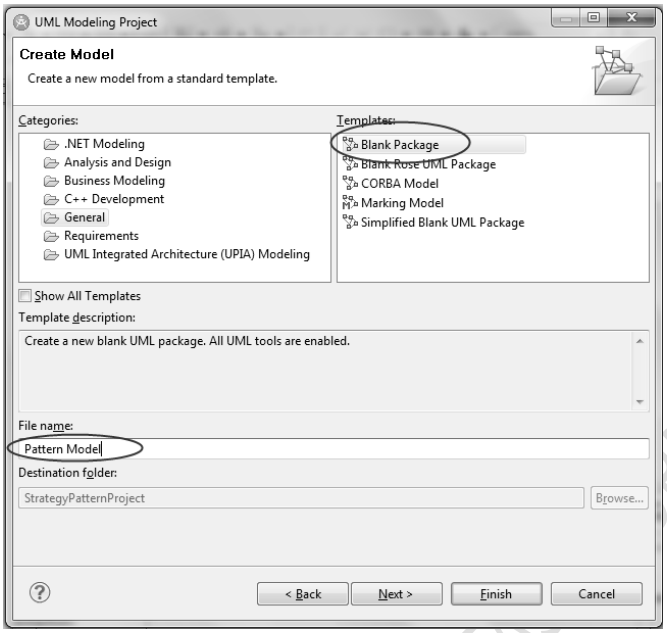
Gambar 8-6 Memilih *UML Project* pada *wizard*

(4) Tuliskan nama proyek pada *textbox "Project name"*, misalnya dengan nama "*StrategyPatternProject*". Lalu klik *Next*.



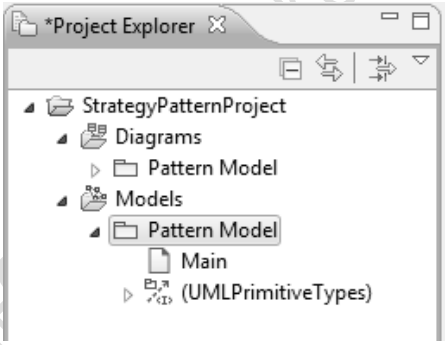
Gambar 8-7 Membuat *Model Project*

(5) Buat model UML dari *template "Blank Package"*, ketikkan nama file dengan "Pattern Model" dan pastikan tipe *diagram* pada *Project Explorer* adalah *FreeFrom Diagram* (ikon Free Form Diagram) sebagai tipe *diagram default*, lalu klik *Finish*.



Gambar 8-8 Membuat *model*

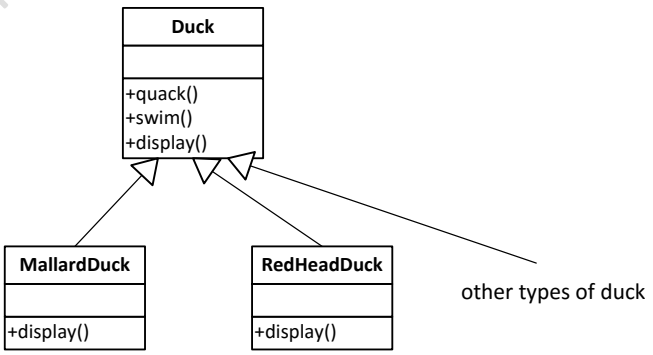
Pada *Project Explorer* akan muncul ACMEPatternProject > Models > PatternModel



Gambar 8-9 Tampilan *UML Project* ACMEPatternProject pada *Project Explorer*

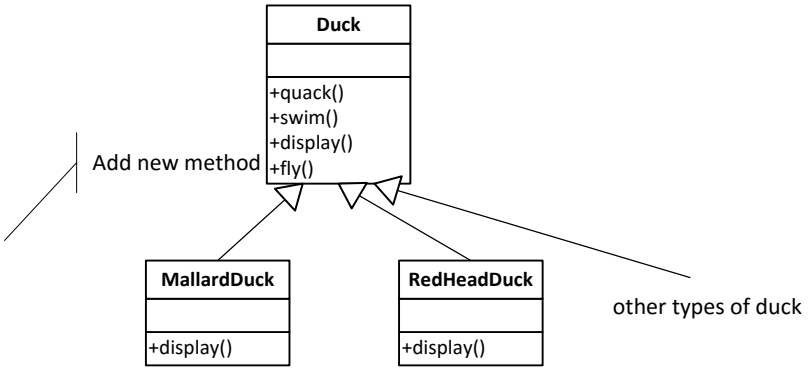
8.3.2 MENGIMPLEMENTASIKAN STRATEGY PATTERN

Perhatikan simulasi bebek dengan *diagram* kelas di bawah ini:



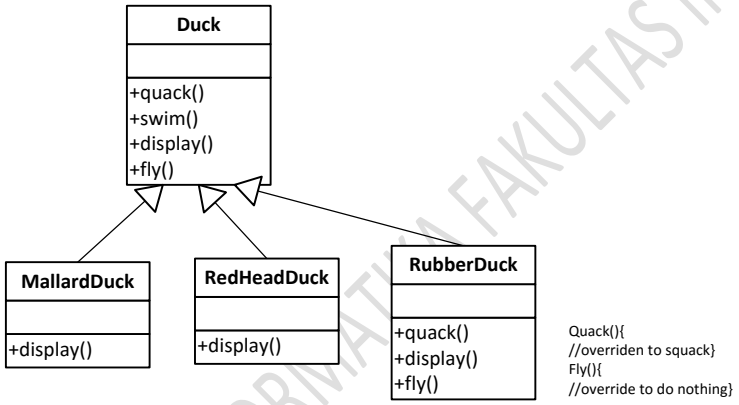
Gambar 8-10 *Duck Simulation* (Kurikulum Program Studi S1 Informatika 2003, 2003)

Bagaimana jika kita ingikan bebek-bebek itu untuk terbang?



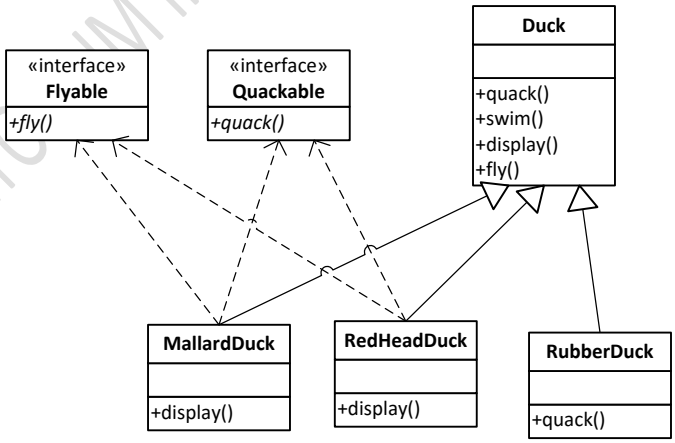
Gambar 8-11 Menambahkan method baru *fly()* di *duck simulation* (Kurikulum Program Studi S1 Informatika 2003, 2003)

Masalahnya, tidak semua bebek bisa terbang, bebek karet *RubberDuck* tidak terbang dan juga tidak bersuara kwek-kwek (Kurikulum Program Studi S1 Informatika 2003, 2003).



Gambar 8-12 *RubberDuck* tidak bisa terbang dan bersuara kwek-kwek (Kurikulum Program Studi S1 Informatika 2003, 2003)

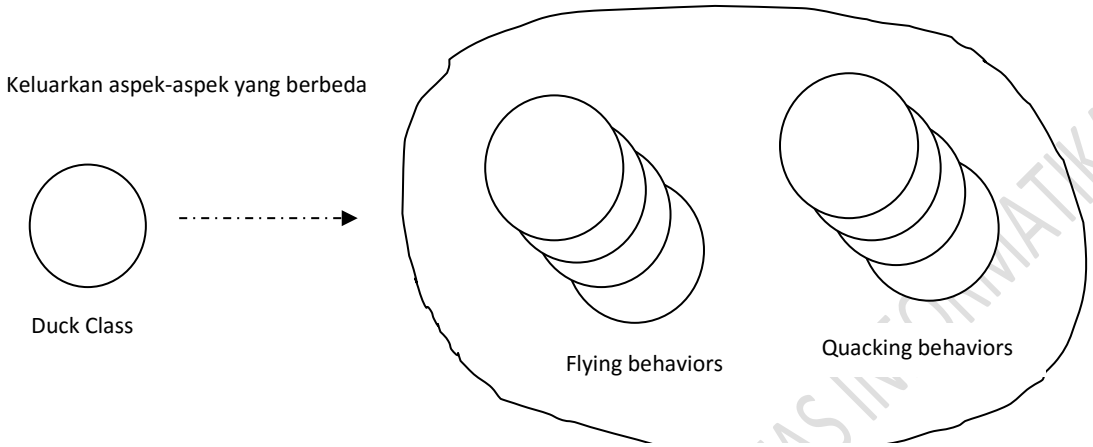
Bagaimana jika kita gunakan *interface?*



Gambar 8-13 Mengimplementasikan *interface* untuk *duck simulation* (Kurikulum Program Studi S1 Informatika 2003, 2003)

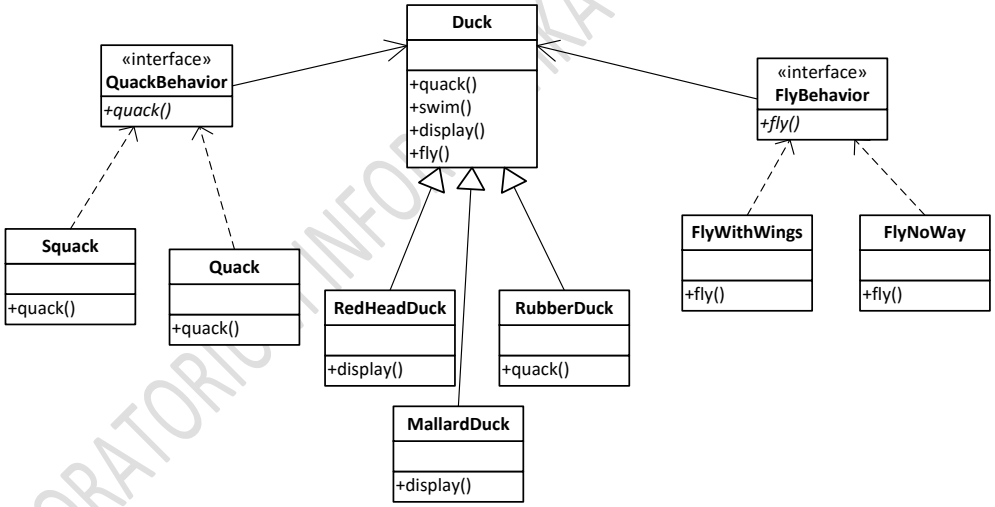
Jika dengan solusi ini, ada kode yang diduplikasi, dimana kelas MallardDuck dan RedHeadDuck harus mengimplementasikan dua *method* fly() dan *quack*().

Prinsip Desain: Identifikasi aspek-aspek aplikasi yang berbeda dan pisahkan aspek tersebut dari yang aspek yang tetap sama (Kurikulum Program Studi S1 Informatika 2003, 2003).



Gambar 8-14 Prinsip desain 1: Memisahkan aspek-aspek yang berubah dari aspek-aspek yang tetap sama (Kurikulum Program Studi S1 Informatika 2003, 2003)

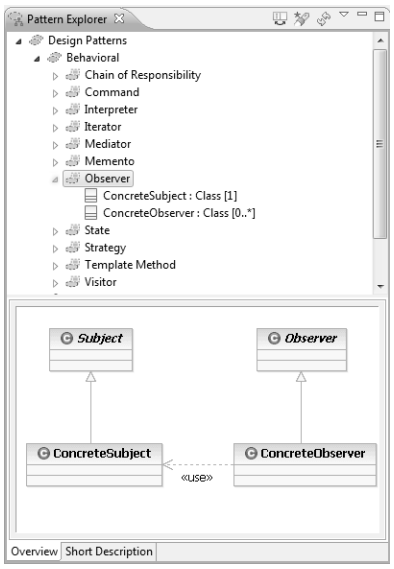
Sehingga kita dapatkan hasil dari penerapan prinsip desain di atas, *diagram* kelas kita menjadi:



Gambar 8-15 *Duck Simulation* dengan *Strategy Pattern* (Kurikulum Program Studi S1 Informatika 2003, 2003)

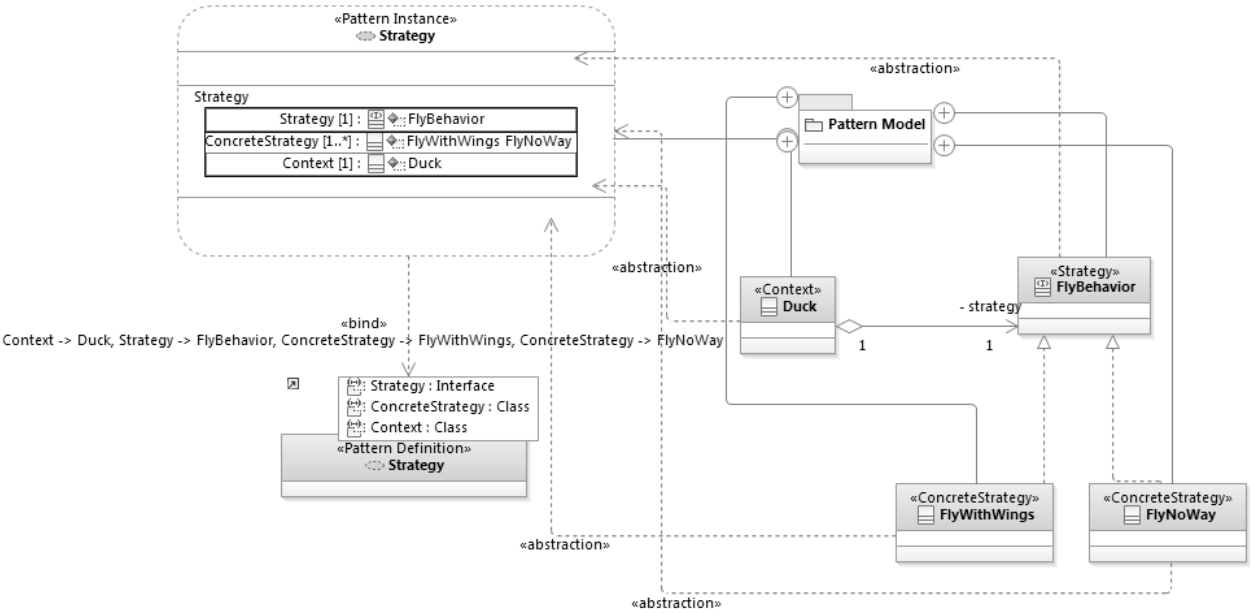
Baru saja diimplementasikan *STRATEGY PATTERN.* Selanjutnya, akan diimplementasikan *Duck Simulation* di atas dengan RSA. Kalimat penghubung ke point 1,2 dst?

1. Pada *Pattern Explorer view*, pilih "*Design Patterns -> Behavioral*". Lalu pilih "*Strategy*".
2. Saat memilih "*Strategy*", frame di bawah *Pattern Explorer* akan menampilkan deskripsi singkat dari *Strategy pattern* (*Short Description*). Pilih tab "Overview" untuk melihat gambaran relasi dari elemen-elemen *pattern*. Gambar berikut menunjukkan *Overview* dari *Strategy pattern.*



Gambar 8-16 Tampilan *Pattern Explorer* dan *Overview* dari *Strategy Pattern*

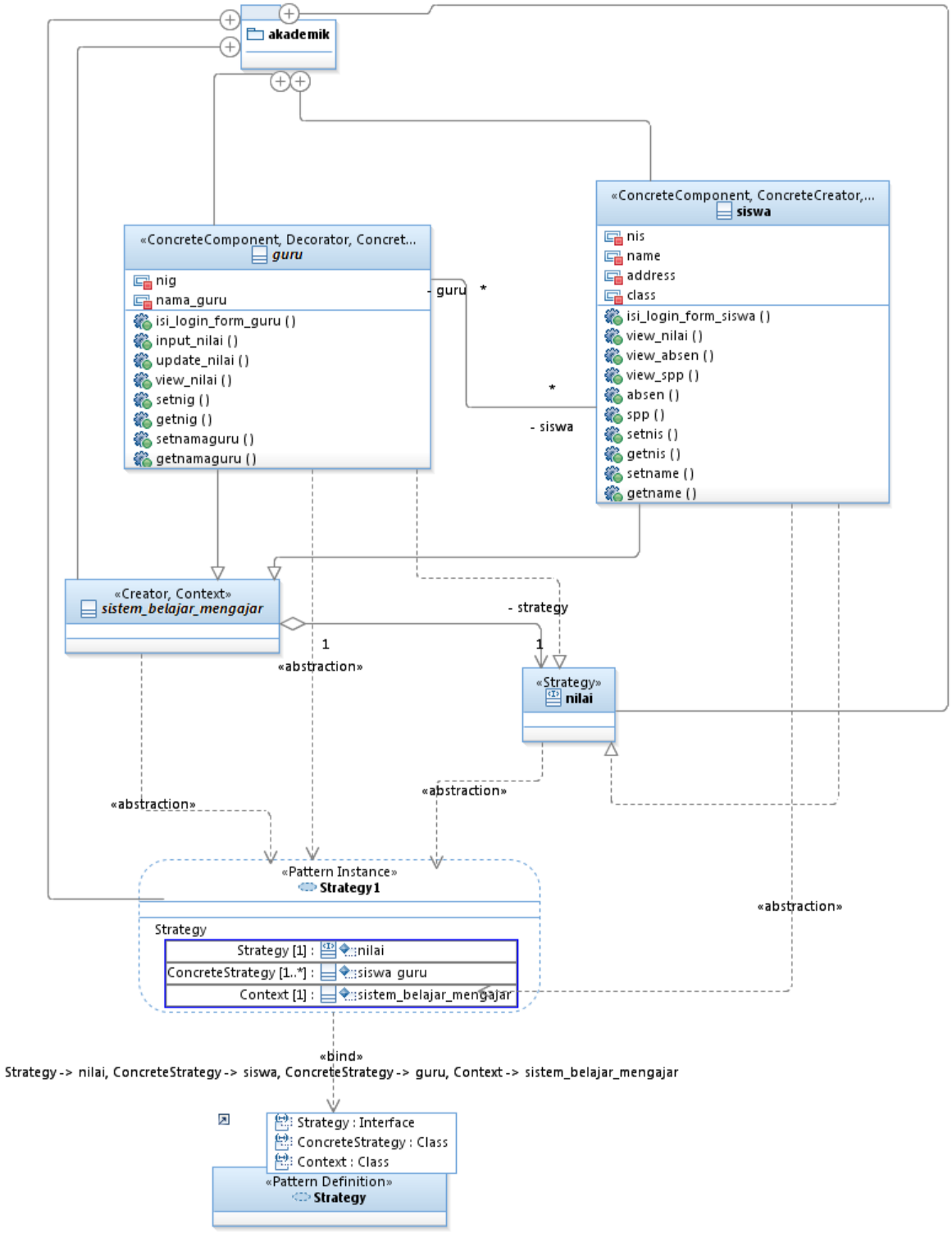
3. Tarik ikon *Strategy* ke *Main diagram*
4. Menghubungkan parameter `Strategy` via *action bar*
   a) *Hover* mouse di atas parameter `Strategy` sehingga *action bar* terlihat.
   b) Klik ikon pada parameter *Context*, ketikkan "Duck" untuk membuat kelas *Duck*.
   c) Periksa kelas UML <<Context>> Duck ada di dalam Pattern Model pada *Pattern Explorer*.
5. Menambahkan *argument* ke parameter Strategy dan ConcreteStrategy
   a) *Hover* mouse di atas parameter Strategy sehingga *action bar* kelihatan
   b) Klik ikon pada paramter *Strategy*, ketikkan "FlyBehavior" di textbox yang dihasilkan
   c) Periksa kelas UML <<Strategy>> FlyBehavior ada di dalam *Pattern Model* pada *Project Explorer* .
   d) Klik ikon pada parameter ConcreteStrategy, ketikkan "FlyWithWings"
6. Ikatkan kelas yang ada dengan parameter ConcreteStrategy
   a) Klik-kanan Pattern Model dan pilih Add UML > Class.
   b) Ketikkan FlyNoWay pada *textbox* yang dihasilkan
   c) Tarik kelas FlyNoWay ke parameter asterisk (*) ConcreteStrategy
   d) Periksa dua *arguments* terlihat pada parameter ConcreteStrategy melalui *Pattern Explorer*
7. Melihat relasi-relasi *pattern*
   a) Klik-kanan pada *Main diagram* dan pilih *Select > All Shapes.*
   b) Klik-kanan elemen model manapun pada *diagram* dan pilih *Filters > Show Related Elements*
   c) Pada daftar "Custom Query", klik *Show All Relationships [Default],* kemudian klik OK.
   d) Periksa kelas-kelas Strategy, ConcreteStrategy dan Context serta relasi-relasi diantara mereka terlihat pada *Main Diagram*



Gambar 8-17 Seluruh elemen yang terhubung satu sama lain

8. Menambahkan *method/operation* pada kelas
   a) *Hover* mouse pada kelas Duck
   b) Ketikkan “performFly” pada *textbox* yang dihasilkan

Contoh Strategy Pattern Akademik:



Gambar 8-18 Strategy Pattern Akademik

MODUL 9 FACTORY METHOD PATTERN

Tujuan Praktikum:

1. Mahasiswa memahami dasar-dasar *structural patterns*
2. Menggunakan *factory method pattern* pada suatu model
3. Menghubungkan elemen-elemen UML dengan parameter-parameter suatu *structural pattern*

9.1 FACTORY METHOD PATTERN

Factory Pattern mendefinisikan sebuah *interface* untuk membuat objek, tetapi membiarkan *subclass* menentukan kelas mana yang akan diinstansiasi. *Factory Method Pattern* memungkinkan sebuah kelas menunda melakukan instansiasi *subclass. Factory Method Pattern* sering juga disebut *Factory Pattern* ataupun *Virtual Constructor* (Gamma, Helm, Johnson, & Vlissides, 1995).

Pattern ini mendefinisikan sebuah *interface* untuk membuat objek tanpa mengetahui kelas objek yang membuatnya. Setiap *Factory Method Pattern* dapat mendefinisikan kelas yang akan dipakai berdasarkan parameter input dan spesifik dari situasi. *Architectural discovery algorithm* mengidentifikasi *pattern* yang terdiri atas kelas *Creator, subclass Concreate Creator,* kelas *interface Product,* dan satu objek *Concreate Object.* Kelas *Creator* menentukan *interface* yang akan membuat kelas *interface Product. Subclass Concreate Creator* mengimplementasikan *interface* ini dengan cara menginstansiasi objek *Concreate Product.* (Help - Rational Software Architect, 2011).

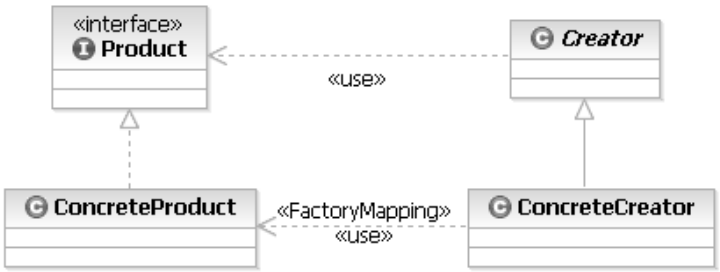
9.2 ELEMEN-ELEMEN FACTORY METHOD PATTERN

Berikut elemen-elemen factory method pattern

- `Product`: Classifier [1] → Menentukan kelas abstrak atau *interface* yang seluruh kelas *ConcreteProduct* mengimplementasikannya
- `ConcreteProduct`: Class [1..*] → merealisasikan kelas *Product* tertentu yang akan dibuat oleh *ConcreatCreator*
- `Creator`: Class [1] → Mendefinisikan suatu kelas abstrak dengan satu *method/*operasi *abstract* yang memasukkan *factory method.*
- `ConcreteCreator` [1..*] → kelas konkrit yang mengimplementasikan *method* kelas *Creator Factory*

9.3 OVERVIEW

Berikut gambaran factory method pattern pada Rational Software Architect



Gambar 9-1 *Overview* dari *Factory Method Pattern* pada *Rational Software Architect*

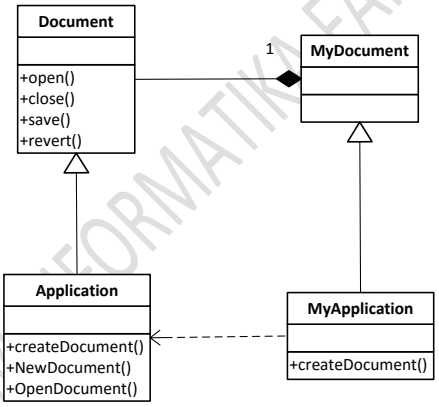
9.4 MENGIMPLEMENTASIKAN FACTORY METHOD PATTERN

Frameworks menggunakan kelas abstrak untuk mendefinisikan dan memelihara relasi antar objek. *Framework* bertanggungjawab untuk membuat objek-objek itu juga.

Bayangkan sebuah *framework* yang dapat menyediakan dokumen kepada pengguna. Dua kelas abstrak utama dalam *framework* ini adalah Kelas *Application* dan kelas *Document.* Kedua kelas abstrak, dan klien harus membuat *subclass* kelas-kelas abstrak tersebut untuk implementasi aplikasi yang spesifik. Untuk membuat aplikasi *drawing,* misalkan, kelas *DrawingApplication* dan *DrawingDocument* dapat didefinisikan. Kelas *Application* betanggungjawab untuk mengatur *Documents* dan akan membuat mereka seperlunya – ketika pengguna memilih *Open* atau *New* dari *Menu*, misalnya.

Karena *subclass* Dokumen tertentu menginstansiasi aplikasi-spesifik, kelas *Application* tidak dapat memprediksi *subclass Document* yang akan menginstansiasinya – kelas *Application* hanya mengetahui ketika *document* baru harus dibuat, bukan jenis kelas *Document* apa yang akan dibuat. Hal ini menghasilkan dilemma: *framework* harus menginstansiasi kelas, tapi yang diketahui hanya kelas-kelas abstrak saja, yang tidak dapat diinstansiasi.

Factory Method Pattern menawarkan solusi. *Pattern* ini mengenkapsulasi pengetahuan *subclass Document* mana yang dibuat dan memindahkan pengetahuan tersebut kepada *framework.*



Gambar 9-2 Framework penyedia dokumen

Untuk menggunakan *factory method pattern* pada Rational Software Architect (RSA), hal yang pertama kali harus dilakukan adalah membuka *Pattern Explorer* , *kemudian* membuat *UML Project* dan *UML Model.*

Cara membuka Pattern Explorer, membuat UML Project dan *UML Model dibahas di Modul 8.*

Ikuti langkah-langkah berikut ini untuk membuat *factory method pattern* pada RSA:

1. Pada *Pattern Explorer view*, pilih "*Design Patterns -> "Creational*". Lalu pilih "*Factory Method*".
2. Tarik ikon *Strategy* ke *Main diagram*
3. Menghubungkan parameter *Factory Method* via *action bar*
   a) Klik ikon  pada parameter *Product*, ketikkan "Document" untuk membuat kelas *Document*.
   b) Periksa kelas UML <<Product>> Document ada di dalam Pattern Model pada *Pattern Explorer*.
4. Menambahkan *argument* ke parameter ConcreteProduct, Creator dan ConcreteCreator
   a) *Hover* mouse di atas parameter ConcreteProduct sehingga *action bar* kelihatan

b) Klik ikon  pada paramter ConcreteProduct, ketikkan “MyDocument” di textbox yang dihasilkan

c) Periksa kelas UML <<ConcreteProduct>> MyDocument ada di dalam *Pattern Model* pada *Project Explorer* .

d) Klik ikon  pada parameter Creator, ketikkan “Application”

e) Klik lagi ikon  pada parameter ConcreteCreator, dan ketikkan “MyApplication”

f) Klik ikon  pada parameter F*actor*yMapping untuk membuat *parameter relationship*

g) Klik *MyDocument* pada daftar *Target Elements*

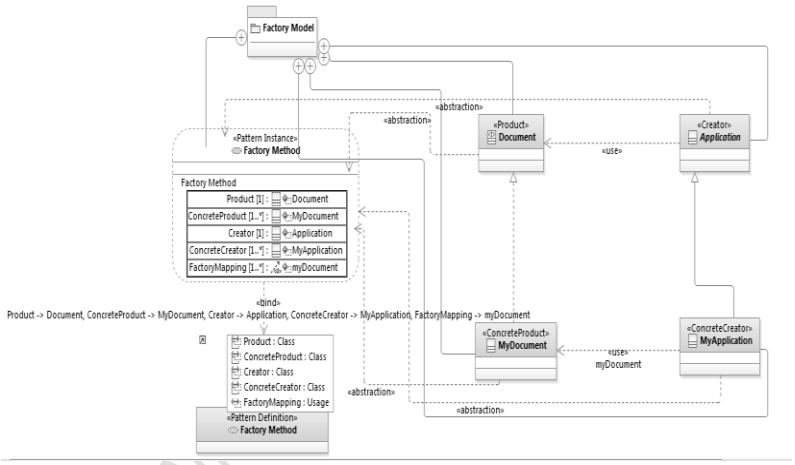
5. Melihat relasi-relasi *pattern*

a) Klik-kanan pada *Main diagram* dan pilih *Select > All Shapes.*

b) Klik-kanan elemen model manapun pada *diagram* dan pilih *Filters > Show Related Elements*

c) Pada daftar "Custom Query", klik *Show All Relationships [Default],* kemudian klik OK.

d) Periksa kelas-kelas Product, ConcreteProduct, Creator dan ConcreteCreator serta relasi-relasi diantara mereka terlihat pada *Main Diagram*



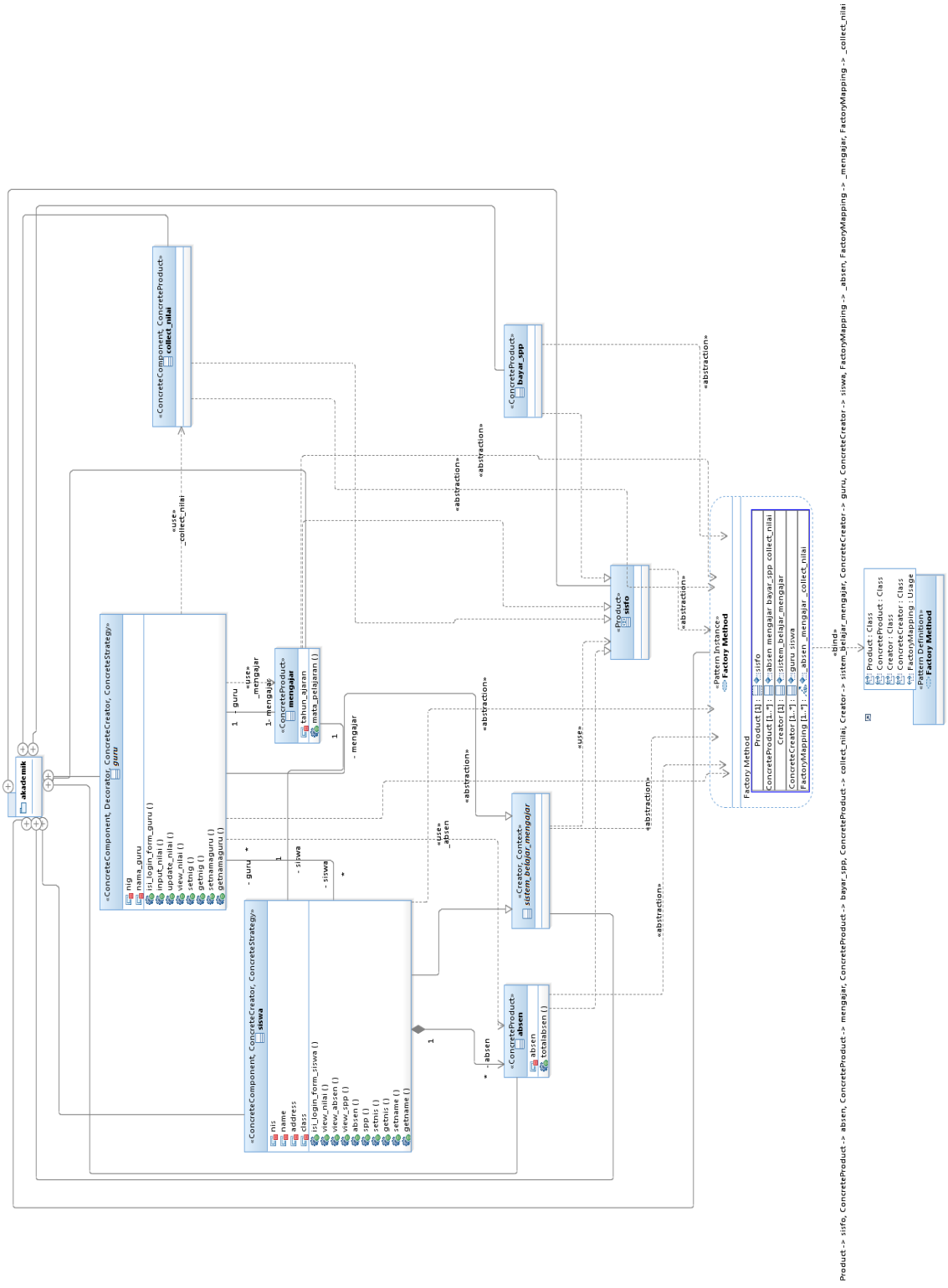
Gambar 9-3 Seluruh elemen yang terhubung satu sama lain

6. Menambahkan *method/operation* pada kelas

a) *Hover* mouse pada kelas Document

b) Ketikkan “performFly” pada *textbox* yang dihasilkan

Contoh Factory Pattern Akademik:



Gambar 9-4 Factory Pattern Akademik

MODUL 10 MENGIMPLEMENTASIKAN DECORATOR PATTERN

Tujuan Praktikum:

1. Mahasiswa memahami dasar-dasar *creational patterns*
2. Menggunakan *decorator* pada suatu model
3. Menghubungkan elemen-elemen UML dengan parameter-parameter suatu *creational pattern*

10.1 DECORATOR PATTERN

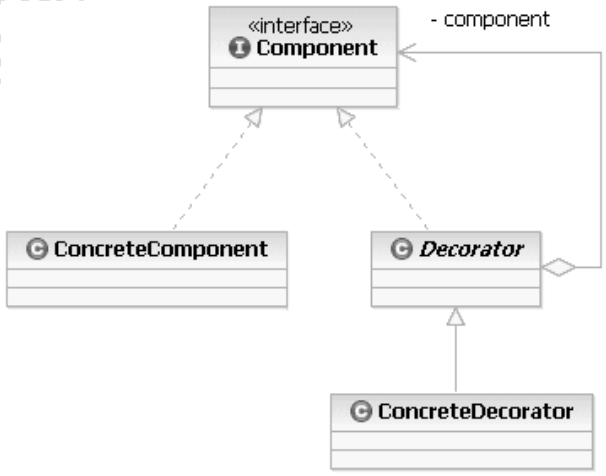
Decorator pattern memungkinkan *behavior* dari suatu objek untuk ditambahkan secara dinamis. *Decorator Pattern* termasuk dalam kategori *structural pattern.*

Pattern ini menambahkan *responsibilities* pada suatu objek secara dinamis, tanpa mengubah *interface*-nya. *Decorator pattern* bekerja seperti pembungkus karena *pattern* ini mengimplementasikan *interface* aslinya, menambah kapabilitas, dan mendelegasikan pekerjaan ke objek aslinya, sehingga kita dapat menggunakannya sebagai alternatif untuk membuat *subclass. Pattern* ini juga dikenal dengan nama *Wrapper.*

10.2 ELEMEN-ELEMEN DECORATOR PATTERN:

Berikut elemen-elemen decorator pattern :

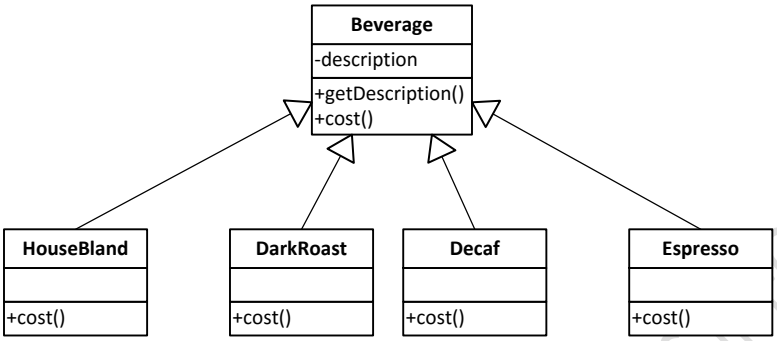
- `Component`: Interface [1] → Menspesifikasikan *interface* yang mengimplementasikan kelas `ConcreteComponent`
- `ConcreteComponent`: Class [1..*] → Satu objek yang merealisasikan *interface* `Component` dan merupakan objek yang akan didekorasi dengan *behavior* tambahan
- `Decorator`: Class [1] → Mendefinisikan kelas dasar dimana dengan kelas dasar tersebut seluruh kelas `ConcreteDecorator` harus menrealisasikannya dengan *behavior* tambahan
- `ConcreteDecorator`: Class [1..*] → Meng-*extend* kelas `Decorator` dasar dengan *behavior* tambahan



Gambar 10-1 *Decorator Pattern*

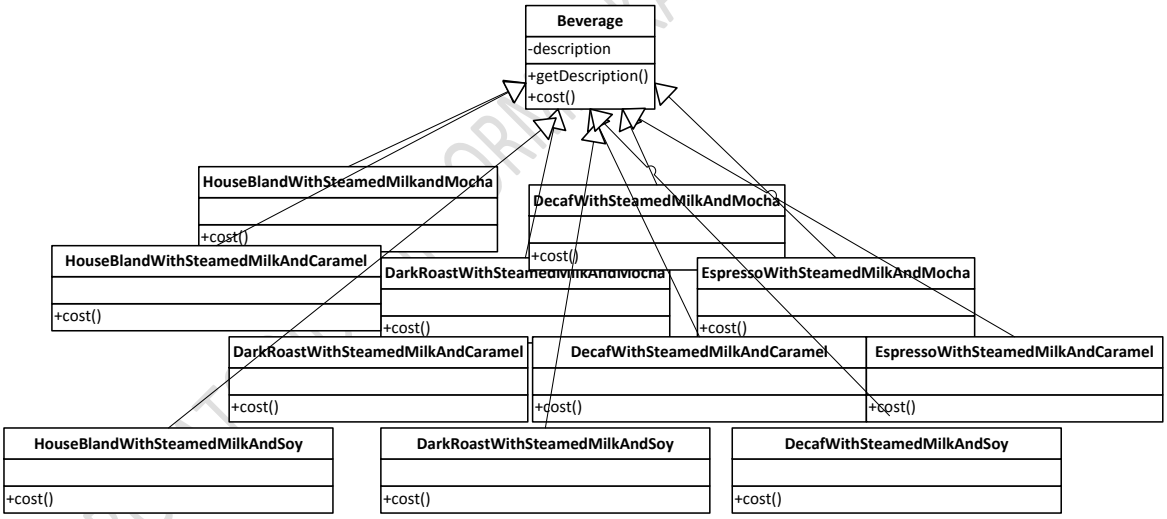
10.3 MENGIMPLEMENTASIKAN DECORATOR PATTERN

Karena perkembangan bisnisnya yang sangat cepat, Starbuzz Coffe membuat sistem pemesanan yang fleksibel menyesuaikan penawaran minuman yang mereka sediakan. Ketika pertama kali membuka bisnisnya, sistem pemesanan yang dirancang seperti ini:



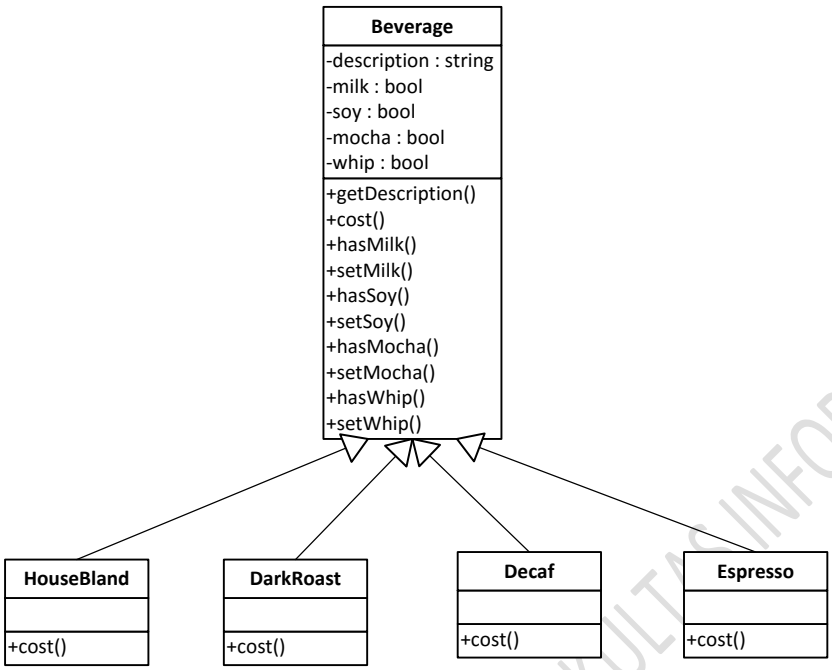
Gambar 10-2 Sistem pemesanan *Starbuzz Coffe*

Sesuai dengan keinginan pelanggan, mereka dapat meminta beberapa bumbu tambahan seperti *steamed milk, soy* dan *mocha,* dan semuanya ditaburkan di atas *whipped milk.* Starbuzz menambahkan sedikit tambahan harga untuk setiap tambahan bumbu yang dipesan, sehingga mereka harus membuat sistem pemesanan mereka menjadi seperti ini:



Gambar 10-3 Sistem pemesanan Starbuzz Coffee untuk menangani tambahan bumbu

Jika seperti ini, tentu akan terjadi ledakan jumlah kelas. Bagaimana jika gunakan variabel instans dan pewarisan (*inheritance*) pada kelas super (*superclass*) untuk mengatasinya? Jika seperti itu, kelas *Beverage* akan diubah dengan menambahkan variabel instans untuk merepresentasikan apakah setiap minuman menambah *milk, soy, mocha* dan *whip.*



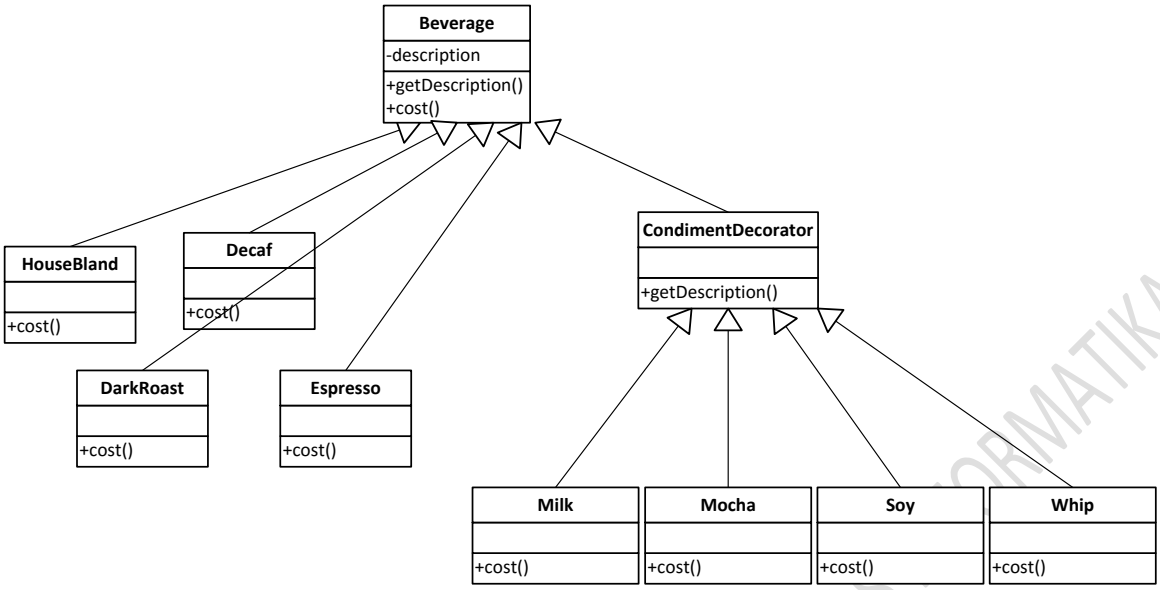
Gambar 10-4 Kelas Beverage dengan menggunakan variabel instans dan *inheritance*

Prinsip Desain: Kelas harus bisa di-*extend,* tapi tertutup untuk dimodifikasi

Desain yang baik mengizinkan kelas-kelasnya untuk di-*extend* oleh kelas lain dengan *behavior* baru yang diinginkan. Jika kebutuhan atau *requirements* berubah, setiap kelas lain dapat membuatnya sesuai kebutuhan masing-masing.

Akan tetapi, karena desain yang baik juga sudah melalui waktu yang panjang untuk menjadikannya baik dan benar dan juga tidak ada *bug,* maka desain yang baik akan tidak mengizinkan perubahan terjadi pada kode yang sudah dibuat. Desain yang baik akan tetap menjaga kelas-kelasnya untuk tidak dimodifikasi. Prinsip ini juga disebut *open-closed principle.*

Decorator pattern memegang teguh prinsip desain ini. *Decorator pattern* memungkinkan *behavior* suatu objek untuk ditambahkan secara dinamis. Bagaimana dengan kasus di atas? Jika implementasi kasus di atas dengan *decorator pattern,* akan didapatkan desain baru seperti berikut ini:



Gambar 10-5 Starbuzz Coffee Ordering System dengan Decorator Pattern

Untuk menggunakan *decorator pattern* pada RSA, hal yang pertama kali harus dilakukan adalah membuka *Pattern Explorer* , *kemudian* membuat *UML Project* dan *UML Model.*

Cara membuka Pattern Explorer, membuat UML Project dan *UML Model dibahas di Modul 8.*

Ikuti langkah-langkah berikut ini untuk membuat *decorator pattern* pada RSA:

1. Buatlah *UML Project* dengan nama "DecoratorPatternProject" dan *UML Model* dengan nama "DecoratorModel"
2. Pada *Pattern Explorer*, pilih "*Design Patterns -> Structural*". Lalu pilih "*Decorator*".
3. Saat memilih "*Decorator*", *frame* di bawahnya akan menampilkan deskripsi singkat tentang *Decorator Pattern* pada *tab Short Description* dan gambaran relasi dari elemen-elemen *pattern* pada tab "Overview".

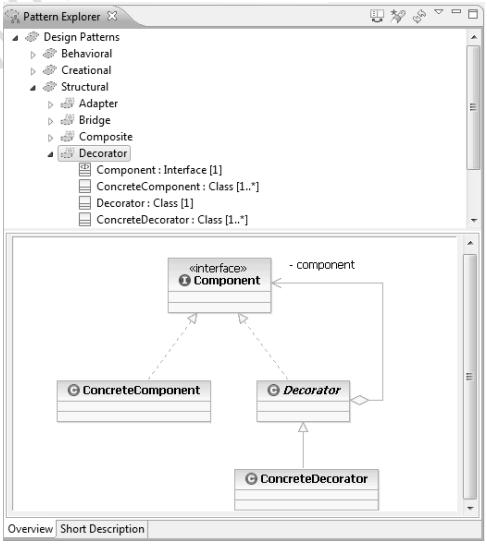
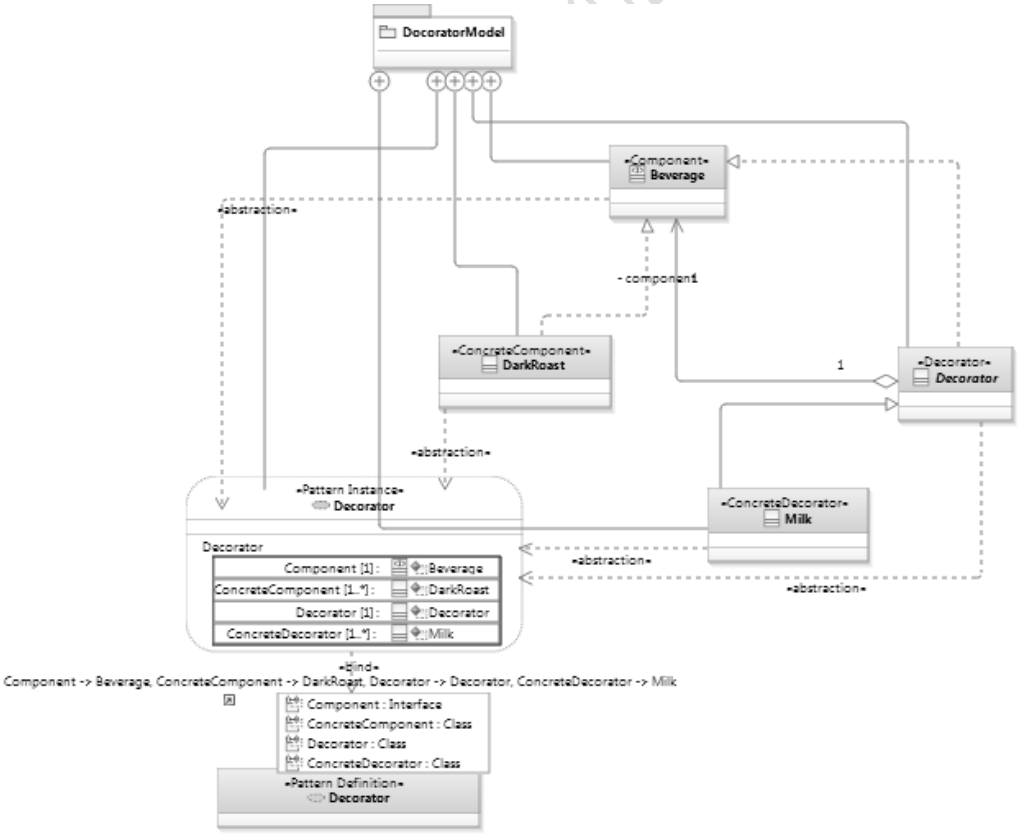


Figure 1 *Decorator pattern* pada *View Pattern Explorer* dan *Overview*-nya

4. *Drag* ikon *Strategy* ke Main *diagram* dari DecoratorModel.

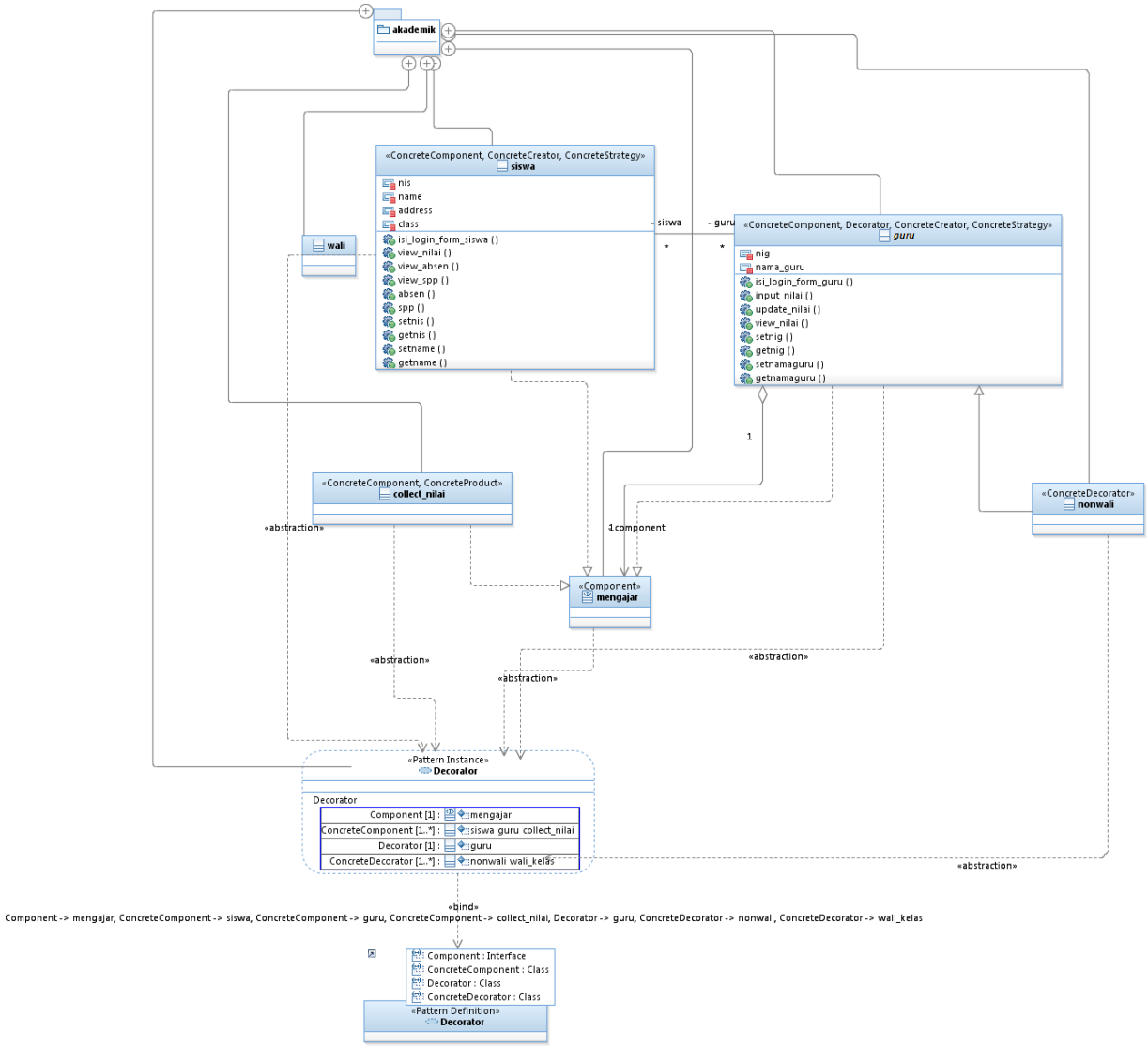
5. Menghubungkan parameter Strategy via *action bar*
   a) Klik ikon , ketikkan “Beverage” untuk membuat kelas *Duck*.
   b) Periksa kelas UML <<Component>> Beverage ada di dalam Decorator Model pada *Pattern Explorer*.
6. Menambahkan *argument* ke parameter ConcreteComponent, Decorator dan ConcreteDecorator
   a) *Hover* mouse di atas parameter ConcreteComponent sehingga *action bar* kelihatan
   b) Klik ikon pada parameter ConcreteComponent, ketikkan “DarkRoast”
   c) Periksa kelas UML << ConcreteComponent >> DarkRoast ada di dalam *Decorator Model* pada *Project Explorer* .
   d) Klik ikon pada parameter Decorator, ketikkan “Decorator”
   e) Klik ikon pada parameter ConcreteDecorator, ketikkan “Milk”
7. Melihat relasi-relasi *pattern*
   a) Klik-kanan pada *Main Diagram* dan pilih *Select > All Shapes.*
   b) Klik-kanan elemen model pada *diagram* dan pilih *Filters > Show Related Elements*
   c) Pada daftar "Custom Query", klik *Show All Relationships [Default],* kemudian klik OK.
   d) Periksa kelas-kelas Component, ConcreteComponent, Decorator dan ConcreteDecorator, serta relasi-relasi diantara mereka terlihat pada *Main Diagram*



Gambar 10-6 Seluruh elemen yang terhubung satu sama lain

8. Menambahkan *method/operation* pada kelas
   a) *Hover* mouse pada kelas Duck
   b) Ketikkan “cost” pada *textbox* yang dihasilkan

Contoh Decorator Pattern Akademik:



Gambar 10-7 Decorator Pattern Akademik

DAFTAR PUSTAKA

2007, I. C. (2007). Essentials of IBM Rational Software Architect v7 RD541/DEV396 February 2007 Student Manual. California: IBM Rational Software.

CS 210: File and Data Structures Using an Introduction to Design Patterns as basis Fall 2006. (2006, September 7). Retrieved February 10, 2013, from http://www.csee.wvu.edu/classes/cs210/Fall_2006/CS_210_Sept_7.ppt

Freeman, E., Robson, E., & Bates, B. (2004). *Head First Design Pattern.* California: O'Reilly Media, Inc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns Elements of Reusable Object-Oriented Software, First Edition.* Massachusetts: Addison-Wesley Pub Co.

Help - Rational Software Architect. (2011). New York: IBM Rational Software Architect.

Kurikulum Program Studi S1 Informatika 2003. (2003). Retrieved February 11, 2013, from http://kur2003.if.itb.ac.id/file/Behavioral%20-%20Strategy%20Pattern.pdf

Kurikulum Program Studi S1 Informatika 2003. (2003). Retrieved February 13, 2013, from http://kur2003.if.itb.ac.id/file/Creational%20-%20Factory%20Pattern.pdf

informatics lab