Implementation of Design Pattern for Wireless Weather System Application Design

Penerapan Pola Desain untuk Perancangan Aplikasi Stasiun Cuaca Nirkabel

Lintang Dwi Febridiani

Pusat Penelitian Informatika
Lembaga Ilmu Pengetahuan Indonesia
Gedung 20 lt 3, Jln Sangkuriang 154 D, Bandung
Indonesia

Abstract

One of the problems in object oriented application development is how to achieve a good quality software design. This paper presents a good quality software design for wireless weather system application based on design patterns. Abstract Factory and Mediator design patterns are used to design the application based on problem similarity. Design process produces a static and a dynamic model of the application in UML (Unified Modeling Language) diagrams. We use class cohesion and coupling metrices to measure the quality of our design. Measurements show our proposed design has high class cohesion and low class coupling.

Key Words: object oriented design, design pattern, unified modeling language

Abstrak

Salah satu permasalahan dalam pengembangan aplikasi berbasis objek adalah bagaimana mendapatkan desain aplikasi yang berkualitas. Makalah ini mempresentasikan sebuah perancangan aplikasi stasiun cuaca nirkabel yang berkualitas berdasarkan pola desain (*design pattern*). Pola desain *Abstract Factory* dan *Mediator* digunakan pada perancangan berdasarkan kesamaan permasalahan desain. Hasil desain adalah desain statis dan dinamis aplikasi yang dimodelkan dalam diagram-diagram UML (*Unified Modeling Language*). Ukuran kohesi dan *coupling* kelas digunakan untuk menilai kualitas desain. Hasil pengukuran menunjukkan desain yang diajukan memiliki kohesi kelas yang tinggi dan *class coupling* yang rendah.

Kata kunci: desain berbasis objek, pola desain, *unified modeling language*

1. PENDAHULUAN

Tahap desain pada rekayasa perangkat lunak merupakan tahap penting yang menentukan kualitas suatu perangkat lunak. Jika desain yang dibuat memiliki kualitas yang baik maka sebuah perangkat lunak akan juga memiliki kualitas yang baik yang dicirikan oleh: fungsionalitas, realibilitas, usabilitas, efesiensi, *maintainability*, dan *portability* [1].

Dalam desain perangkat lunak, bagian sebuah aplikasi yang dibuat umunya bukan permasalahan yang benar-benar baru. Permasalahan desain yang dihadapi umumnya menyerupai permasalahan lain yang bertujuan sama. Memang tidak persis sama, namun memiliki pola yang sama. Permasalah desain yang umum itu biasanya disertai dengan solusi-solusi desain perangkat lunak. Solusi yang sudah teruji tersebut dikenal dengan pola desain (*design pattern*) [2]. Keuntungan penggunaan *design pattern* pada desain perangkat lunak adalah mempersingkat waktu desain dan mendapatkan desain perangkat lunak yang baik.

Penerapan pola desain disesuaikan dengan permintaan perangkat lunak. Pada kasus antar muka aplikasi stasiun cuaca berbasis `Qt` diperlukan analisis pemilihan jenis pola desain dan penerapannya yang tepat. Pada [3] telah dipresentasikan implementasi antar muka grafis untuk stasiun cuaca nirkabel berbasis `Qt` namun desain aplikasinya masih mencampur antara proses bisnis, model entitas dan antarmuka sehingga akan

*Corresponding Author. Tel: +6222-2504711
Email: lintang@informatika.lipi.go.id
Received: 7 Oct 2012; revised: 7 Nov 2012; accepted: 19 Nov 2012
Published online: 26 Nov 2012

susah mendapatkan ciri perangkat lunak yang berkualitas.

Sebuah penerapan *design pattern* untuk mengatasi permasalahan kualitas perangkat lunak pada [3] dipresentasikan oleh makalah ini. Dimulai dengan pemilihan jenis pola desain yang tepat, modifikasi pola desain, dan menghasilkan model statis dan dinamis aplikasi stasiun cuaca nirkabel. Model statis dan dinamis aplikasi dibuat dalam diagram pada UML. Diagram-diagram UML ini sangat mudah diterjemahkan menjadi kerangka kode pada implementasi aplikasi cuaca nirkabel berbasis `Qt`.

Pola desain pertama kali dikenalkan dan dibukukan oleh Gamma et al (*Gang of Four*) pada tahun 1999 [2] yang berisi kumpulan desain perangkat lunak berbasis objek yang berkualitas ”baik”. Sejak saat itu *design pattern* telah dikaji dan digunakan seperti: [4] mengaplikasikan dan merumuskan pola desain untuk pengembangan aplikasi *game*, [5] mengusulkan perancangan sistem *embedded* dengan menggunakan pola desain, dan [6] mengkaji kualitas rancangan perangkat lunak yang menggunakan pola desain. Penggunaan *design pattern* untuk rancangan antar muka dapat ditemukan pada [7] dan [8]. [7] membuat rancangan antar muka untuk aplikasi bergerak dengan menggunakan pola desain. Sedangkan [8] mengusulkan model rekayasa antarmuka pengguna dengan pola desain untuk aplikasi umum. Makalah ini berbeda dengan [7] dan [8] karena diterapkan pada sistem *embedded* dan untuk aplikasi khusus yaitu aplikasi cuaca nirkabel.

Organisasi makalah adalah sebagai berikut: pada bagian 2 akan dibahas konsep pola desain dan *framework* `Qt`. Bagian 3 akan dipresentasikan metodologi penelitian yang dipakai dengan menyajikan cara mengukur kualitas desain perangkat lunak. Bagian 4 menampilkan diagram kelas dan sekuens (model statis dan dinamis) yang diusulkan. Sedangkan bagian 5 akan membahas kualitas desain yang dihasilkan. Makalah ditutup dengan kesimpulan pada bagian 6.

2. POLA DESAIN DAN QT *FRAMEWORK*

2.1 Pola Desain

Pola desain adalah deskripsi tentang kelas dan objek-objek yang berkomunikasi, yang dibuat untuk menyelesaikan persoalan perancangan umum pada konteks tertentu. *Design pattern* adalah sebuah solusi terhadap masalah-masalah umum dalam rekayasa perangkat lunak yang dapat digunakan berulang kali. Pola desain ini merupakan sebuah *template* yang menunjukkan bagaimana sebuah masalah diselesaikan dan dapat digunakan kembali dalam situasi yang berbeda. Pada pemrograman berbasis objek, pola desain ini biasanya menunjukkan hubungan antar kelas (model statis) dan interaksi antar objek (model dinamis). Namun berbeda dengan algoritma, pola desain tidak menunjukkan kelas dan objek yang terlibat pada komputasi [1].

Secara umum, sebuah pola desain memiliki 4 elemen penting, yaitu [1]:

(1) *name*, mendeskripsikan sebuah masalah, penyelesaian dan konsekuensinya dengan 1-2 kata yang mudah dimengerti
(2) *problem*, menjelaskan pada konteks seperti apa sebuah pola digunakan. Meliputi prasyarat yang harus dipenuhi untuk menerapkan sebuah pola
(3) *solution*, mendeskripsikan kelas-kelas yang terlibat pada solusi umum. Juga memperlihatkan elemen-elemen yang membangung sebuah rancangan, seperti apa relasinya, apa tanggungjawabnya dan bagaiamana kolaborasinya.
(4) *consequences*, adalah hasil dan konsekuensi dari implementasi pola dan strategi perancangan yang harus dipertimbangkan

Ada banyak sekali pola desain yang dapat diimplementasikan sesuai dengan kebutuhan, namun yang cukup populer adalah 23 macam pola desain yang dikenalkan oleh The Gang of Four (GoF). Secara umum, ke-23 pola tersebut dibagi menjadi 3 kategori berdasarkan fungsinya, yaitu [1]:

(1) *Creational Patterns*, yaitu pola yang fokus pada inisiasi kelas/objek;
(2) *Structural Patterns*, yaitu pola yang memberikan komposisi sebuah kelas/objek; dan
(3) *Behavioral Patterns*, yaitu pola yang mengatur tingkah laku, interaksi atau fungsi dari suatu kelas/objek.

2.2 Qt *framework*

`Qt` sangat memungkinkan untuk penerapan pola desain. `Qt` memiliki `Qobject` sebagai kelas statis yang utama, konsep *Signal*, *Slots*, *template* dan *containers*. `Qt` memiliki komponen-komponen yang mendukung penerapan pola desain. *Signal* dan *Slots* mendukung pemrograman berbasis komponen. Komponen dapat menentukan signal yang dikeluarkan pada beberapa kondisi, dan juga parameter-parameternya. Komponen juga dapat menentukan *slot*, yang merupakan metode `C++` standar dengan ditandai khusus sehingga dapat menjadi *slot*. *Signal* dan *Slots* tersebut merupakan bagian dari metode `Qobject::connect()` yang merupakan metode utama dalam `Qt` [9]. Komponen-komponen yang digunakan berulang dapat berupa sebagai berikut : *class*, *namespace*, *header file* (`*.h`), *source code module* (`*.cpp`), *compiled*

object module (`*.o` atau `*.obj`), *library* (`*.lib` atau `*.la`), *devel package* (lib+header files) dan *application* [10].

Template memungkinkan C++ untuk men-*generate* kelas dan fungsi yang berbeda versi dengan tingkah laku yang sama dan tipe yang berparameter. *Template* dibedakan dengan penggunaan kata kunci `template` dan parameter template ditulis dalam kurva sudut <>. *Template* fungsi (function template) digunakan untuk membuat fungsi *type-checked* yang memiliki kesamaan pola. Sedangkan template kelas (*class template*) digunakan untuk membangkitkan *containers* data umum [10]. Sedangkan *containers* adalah kelas untuk mengumpulkan tipe value (yang dapat diperbanyak). Masing-masing struktur data dioptimalkan untuk operasi yang berbeda. Pada `Qt` 4, ada beberapa kelas template containters, diantaranya `Qlist<T>`, `QstringList`, `QlinkedList`, `Qvector<T>`, `Qmap <Key, T>`, `Qstack`, dan lain-lain [10].

Sebagai contoh penerapan pola desain pada `Qt` adalah penerapan pola desain *Mediator*. Tujuan pola desain*Mediator* adalah membuat objek yang mengenkapsulasi satu himpunan objek yang saling berinteraksi. *Mediator* memungkinkan pengembang mengatur interaksi objek sesuai kebutuhan. *Mediator* dapat menjembatani interaksi beberapa elemen seperti *button*, *entry field*, dan *listbox*. Pada `Qt`, yang berperan sebagai *mediator* adalah kelas `Qobject` [10]. Pola *Mediator* juga melibatkan *Colleagues*, kelas antara mediasi yang terjadi dan objek antar muka. Pada `Qt`, kelas ini bisa berupa kelas apa saja yang merupakan turunan langsung maupun tidak langsung dari `Qobject`, sehingga dapat terlibat pada mekanisme *signal/slot*.

3. METODOLOGI

Metodologi penerapan pola desain untuk aplikasi cuaca nirkabel tersusun oleh beberapa langkah sebagai berikut:

(1) **Analisa masalah desain**. Perumusan masalah desain yang tepat akan menentukan tipe *design pattern* yang dipilih. Dalam hal aplikasi cuaca nirkabel terdapat beberapa permasalahan yaitu: (1) Entitas apa saja yang ada?, (2) Bagaimana merepresentasikan entitas?, (3) Apa hubungan struktural antar entitas? dan (4) Apa hubungan tingkah laku antar entitas?

(2) **Pemilihan pola desain**. Pada sebuah *design pattern*, setiap entitas yang terlibat ditentukan perannya dalam desain perangkat lunak.

(3) **Pemodelan struktural**. Model struktural merupakan hasil desain pertama yang menggambarkan hubungan statis antar entitas dalam bentuk diagram kelas.

(4) **Pemodelan tingkah-laku**. Model tingkah laku menggambarkan hubungan pemanggilan operasi entitas oleh entitas lain biasanya dalam rangka memenuhi suatu *use case*.

(5) **Penilaian kualitas desain**. Kualitas desain perangkat lunak dapat dinilai dari aspek kohesifitas pada kelas dan *coupling* antar kelas.

4. DESAIN APLIKASI YANG DIAJUKAN

Berdasarkan metodologi yang dijelaskan pada Bagian 3 beberapa hal yang akan dijelaskan pada bagian ini adalah: (1) hasil analisis entitas yang terlibat, (2) *design pattern* yang dipilih, (3) model struktural yang diajukan dan (4) model *behavioural* yang diajukan.

4.1 Analisis Entitas

Berdasarkan aplikasi cuaca nirkabel yang dijelaskan pada [3] terdapat 1 skenario *use case* yaitu *use case* **tampilkan data terkini** seperti yang dipresentasikan oleh Tabel I. Aktor untuk *use case* ini masih disederhanakan yaitu semua pengguna yang mengunjungi aplikasi sistem cuaca nirkabel. Terdapat 2 alir skenario yaitu: alir utama dan alir *exception*. Alir utama dijalankan apabila pembacaan *string* data berhasil. Selain itu, yang dijalankan adalah alir *exception*

Berdasarkan analisis entitas pada skenario *usecase* menghasilkan beberapa kelas yang ada pada aplikasi cuaca seperti yang ditunjukkan oleh Tabel II.

4.2 Pola Desain

Berdasarkan analisa entitas maka permasalahan desain pada aplikasi ini adalah: (1) Bagaimana merepresentasikan `SensorMgr`, `SensorData` dan `SensorImgGen` untuk tiap-tiap jenis sensor dan (2) Bagaimana agar kelas `MainWindow` tidak memiliki *coupling* yang besar terhadap kelas-kelas yang dipakainya seperti `Parser` dan `SensorMgr`. Representasi `SensorMgr`, `SensorData` dan `SensorImgGen` harus membuat kelas yang menggunakannya tidak perlu merujuk pada jenis sensor karena metode yang dipanggil adalah sama. Sedangkan, `MainWindow` harus sekecil mungkin tersambung (*coupling*) dengan kelas-kelas yang dibuat.

Pemilihan pola desain didasarkan pada kesamaan permasalahan disan yaitu dengan memperhatikan pernyataan *problem* pada pola desain. Berdasarkan itu, dipilih 2 pola desain yang dipakai pada desain aplikasi yaitu *Abstract Factory* dan *Mediator*. Intisari pola desain *Abstract Factory* dan *Mediator* adalah sebagai berikut [2]:

Tabel I. Skenario *use case* **tampilkan data cuaca terkini**

Pra syarat	Data cuaca tersedia dalam format *string* yang dapat diakses melalui protokol `http`	
Kondisi akhir	Data cuaca tertampil pada komponen antarmuka dan tersimpan pada media penyimpanan	
Aktor	Semua pengguna (*User*)	
Alir utama	No	Aksi
	1	*User* membuka aplikasi cuaca
	2	Aplikasi mengambil string data cuaca terkini melalui `http`
	3	*Parser* mem*parsing* string data cuaca
	4	*Parser* memberikan data *humidity*, *windspeed*, *winddirection*, *rainfall*, *airpressure* dan *sunradiation* ke masing-masing *SensorManager*.
	5	Masing-masing *SensorManager* mambangkitkan gambar tampilan data terkini melalui *ImageGenerator*.
	6	*Display* menampilkan gambar tampilan semua data cuaca melalui *SensorManager*
Alir *exception*	No	Aksi
	1	*User* membuka aplikasi cuaca
	2	Aplikasi gagal mengambil string data cuaca terkini melalui `http`
	3	Aplikasi mengambil gambar *null* dari *SensorManager*
	4	Aplikasi menampilkan *dialog exception* data tidak tersedia

(1) *Abstract Factory*.
   (a) ***Problem***: bagaimana membuat *interface* untuk sebuah keluarga objek yang saling berelasi, tanpa secara eksplisit menspesifikasi kelas.
   (b) ***Participants***.
   —`AbstractFactory`: mendeklarasikan *interface* untuk pembuatan objek.
   —`ConcreteFactory`: mengimplementasikan `AbstractFactory` untuk membuat objek.
   —`AbstractProduct`: mendeklarasikan *interface* untuk kelas/tipe objek.
   —`ConcretePoduct`: mendefinisikan objek yang dibuat oleh `ConcreteFactory` dan mengimplementasikan `AbstractProduct`.

Tabel II. Daftar *Class*

Nama Kelas	Keterangan
`MainWindow`	`MainWindow` merepresentasi aplikasi yang merupakan *display* data cuaca
`Parser`	`Parser` merepresentasikan entitas yang dapat melakukan *parsing* terhadap *string* data kiriman sensor-sensor
`SensorManager`	`SensorMgr` merepresentasikan entitas yang berurusan dengan penampilan, dan penyimpanan data sensor terkini. Masing-masing jenis sensor memiliki kelas `SensorMgr` sendiri yaitu: `HumiditySensorMgr`, `WindSensorMgr`, `RainfallSensorMgr`, `AirpressureSensorMgr` dan `SunradiationSensorMgr`
`SensorData`	`SensorData` merepresentasikan entitas yang menyimpan tipe dan format data sensor. Sama seperti kelas `SensorMgr` masing-masing jenis sensor memiliki kelas `SensorData`.
`SensorImgGen`	`SensorImgGen` merepresentasikan entitas yang dapat membangkitkan *image* untuk penampilan data sensor. Sama seperti kelas `SensorMgr` masing-masing jenis sensor memiliki kelas `SensorImgGen`.

(2) *Mediator*.
   (a) ***Problem***: bagaimana membuat objek yang menenkapsulasi sehimpunan objek yang saling berinteraksi.
   (b) ***Participants***.
   —`Mediator`: mendefinisikan *interface* agar objek `Colleague` bisa saling berinteraksi.
   —`ConcreteMediator`: mengimplementasikan *interface* `Mediator` dan mengkoordinasikan komunikasi antara objek `Colleague`.
   —`Colleague`: berkomunikasi dengan `Colleague` lain melalui `Mediator`

4.3 Desain Struktural

Hasil desain struktural aplikasi adalah kelas diagram seperti pada Gambar 1. Gambar 1 merupakan sebagian hasil desain struktural yang berkaitan dengan pembuatan dan penggunaan kelas `SensorManager`.Pola desain *Abstract Factory* digunakan dalam desain struktural

untuk mengenkapsulasi kelas-kelas turunan `SensorManager`.

Pemetaan *participant* pola desain *Abstract Factory* pada diagram kelas Gambar 1 adalah sebagai berikut:

—`AbstractFactory`: `SensorMgrFactory`

—`AbstractProduct`: `SensorMgr`

—`ConcreteProduct`: `HumiditySensorMgr`, `WindSensorMgr`,`RainfallSensorMgr`, `AirpressureSensorMgr` dan `SunradiationSensorMgr`.

Kelas `SensorMgrFactory` dan `SensorMgr` merupakan kelas abstrak/*interface*. Kelas `SensorMgrFactory` berisi satu operasi *static* (operasi yang bisa dieksekusi tanpa perlu ada objek) untuk membuat objek `SensorMgr`. Sedangkan `SensorMgr` berisi *interface* operasi yang perlu diimplementasikan oleh kelas turunannya yaitu operasi `setSensorData` dan `getSensorImage`. Selain itu, terdapat dua definisi kelas utilitas untuk `SensorMgr` yaitu kelas `SensorData` untuk merepresentasikan satu data bacaan sensor dan `SensorImgGenerator` untuk merepresentasikan kelas yang mengandung operasi membangkitan citra data sensor.

Keunggulan desain kelas pada Gambar 1 adalah pengguna paket `SensorMgr` yaitu `WWSMediator` hanya berurusan dengan `SensorMgr` dan `SensorMgrFactory` tanpa harus langsung menggunakan kelas-kelas kongkrit. Hal ini menunjukkan tercapainya enkapsulasi pada aras paket. Selain itu, jika ada penambahan jenis sensor maka desain yang berubah hanya pada kelas konkrit yaitu dengan membuat kelas turunan `SensorMgr` baru misalnya dengan membuat kelas `AltitudeSensor` yang mengimplementasikan/meng-*extend* kelas `SensorMgr`.

4.4 Desain *Behavioral*

Diagram sekuens pada Gambar 2 merupakan hasil desain *behavioral*/dinamis terhadap aplikasi. Desain *behavioral* aplikasi menggunakan pola disain `Mediator` dengan tujuan mengurangi derajat *coupling* antara kelas utama (`MainWindow`) dan kelas model yaitu kelas `Parsing` dan `SensorMgr`.

Gambar 2 berdasarkan interpretasi skenario *use case* yang diberikan oleh Tabel I. Pengguna/*User* membuat objek `MainWindow` yang merupakan kelas utama yang menggandung fungsi `main` dan menciptakan objek *window* untuk *User*. Kelas `WWSMediator` memerankan peran *mediator* untuk beberapa kelas *Colleague* yaitu: `MainWindow`, `Parser` dan `SensorMgr`. Kelas `Parser` merupakan kelas yang mengandung fungsi `parse` yang melakukan *parsing* terhadap *string* kiriman *server* menjadi beberapa data sensor. Sedangkan, kelas `SensorMgr` (yang dibuat oleh kelas `SensorMgrFactory`) mengandung beberapa fungsi yang berurusan dengan penyimpanan data sensor terkini dan perepresentasian data sensor.

5. KUALITAS DESAIN

Untuk menilai kualitas desain aplikasi berbasis objek digunakan 2 kriteria yang umum dipakai yaitu: *class cohesion* dan *class coupling*. *Class cohesion* didefinisikan sebagai ketersatuan elemen pada kelas (tidak termasuk elemen yang diturunkan). *Class cohesion* dinilai rendah (kelas tidak utuh dan bisa dipisahkan) bila kelas merepresentasikan beberapa abstraksi data yang tidak berhubungan, [11]. Sedangkan *class coupling* adalah ukuran seberapa tergantung sebuah kelas dengan kelas-kelas lain [11]. *Class coupling* pada desain yang baik memiliki ukuran yang rendah karena menunjukan modularitas.

5.1 Kohesi Kelas

Salah satu metrik kohesi kelas yang sudah lama diterima adalah `LCOM` (*Lack of Cohesion in Methods*) [11]. `LCOM` dihitung dengan cara:

$$\texttt{LCOM} = \begin{cases} |P| - |Q| & \text{if } |P| < |Q| \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

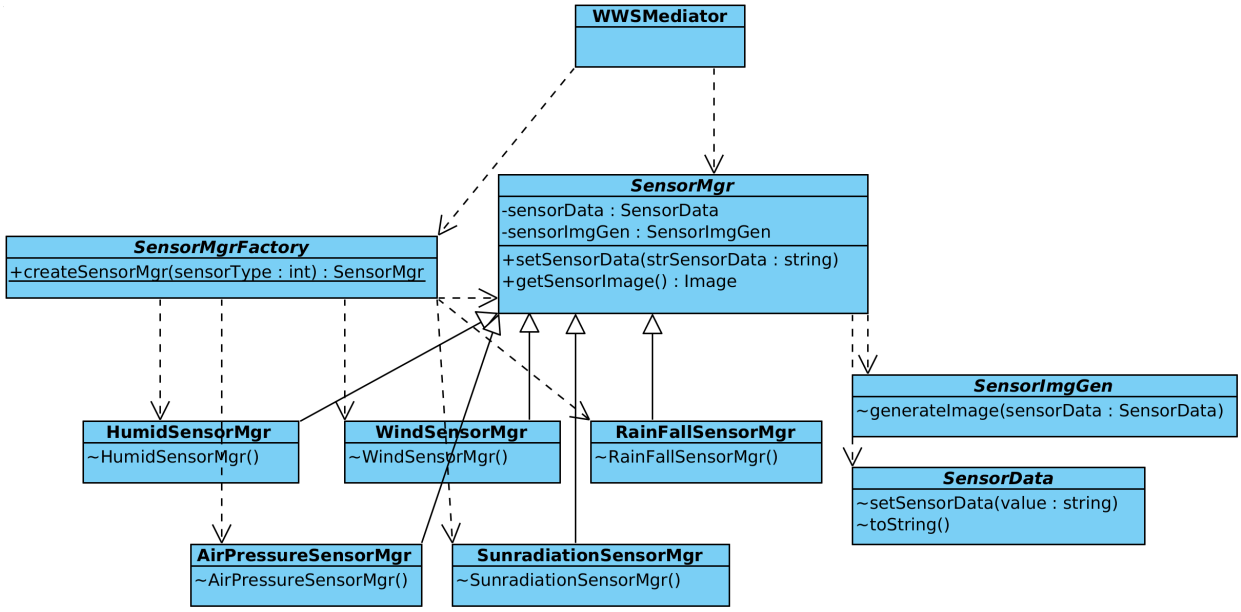
dengan P dan Q adalah:

$$P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\} \quad (2)$$

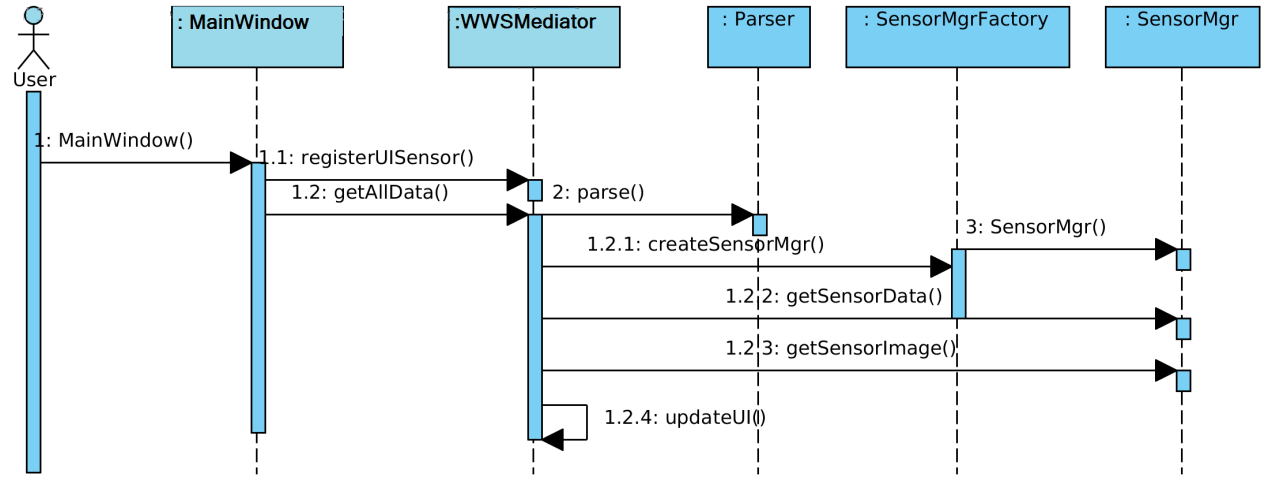
$$Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\} \quad (3)$$

dengan $\{I_i\}$ adalah himpunan objek yang digunakan oleh metode M_i. Sebuah kelas C_1 diasumsikan memiliki metode $M_1, M_2, \cdots, M_n$. Nilai `LCOM` yang rendah dapat disimpulkan secara sementara bahwa kelas memiliki kohesi yang tinggi, sedangkan nilai `LCOM` yang tinggi menunjukkan nilai kohesi yang rendah (kelas bisa dipisahkan).

Nilai `LCOM` untuk kelas `MainWindow` dari [3] adalah 1. Ini menunjukkan kohesi kelas `MainWindow` memiliki kohesi yang cukup. Namun apabila diperhatikan lebih jauh metode $updateData$ pada kelas `MainWindow` melakukan 3 pekerjaan terpisah yaitu: (1) melakukan koneksi ke *Server*, (2) melakukan parsing data dan (3) melakukan tampilan ke objek antarmuka grafis. Fakta ini menunjukkan kohesi metode $updateData$ rendah dan menyebabkan kohesi kelas `MainWindow` juga rendah.



Gambar 1. Desain Struktural `SensorManager` dengan *Abstract Factory*



Gambar 2. Desain *Behavioral* dengan Pola Desain *Mediator*

Sedangkan nilai `LCOM` untuk kelas-kelas yang diusulkan pada desain adalah 0. Hal ini disebabkan oleh tidak ada $I_i \cap I_j = \emptyset$ sehingga $|P| = 0$. Hal ini menunjukkan kelas-kelas yang didesain memiliki nilai kohesi yang tinggi.

5.2 *Class Coupling*

Salah satu metrik *class coupling* yang banyak digunakan adalah *Coupling between object classes* (`CBO`). `CBO` adalah jumlah *coupling* antara satu kelas C_1 dengan kelas lain C_2 yaitu jumlah metode dan jumlah atribut pada C_2 yang digunakan/dipanggil oleh C_1.

Nilai `CBO` pada kelas `MainWindow` pada [3] tidak dapat ditentukan karena hanya mengandung 1 kelas. Sedangkan nilai `CBO` kelas-kelas yang diusulkan diberikan oleh Tabel III.

Tabel III. Nilai `CBO` antar Kelas yang diusulkan

	MW	WM	SF	SM	P	SD	SG
MW	-	2	0	0	0	0	0
WM	2	-	1	2	1	0	0
SF	0	1	-	1	0	0	0
SM	0	2	1	-	0	1	1
P	0	1	0	0	-	0	0
SD	0	0	0	1	0	-	0
SG	0	0	0	1	0	0	-

Berdasarkan Tabel III, nilai `CBO` terbesar adalah 2 yaitu pasangan kelas `MainWindown` (`MW`) dan

kelas `WWSMediator` (`WM`) dan pasangan kelas `WWSMediator` (`WM`) dan kelas `SensorMgr`. Namun, memang kelas `WWSMediator` dirancang sebagai kelas penghubung sehingga nilai *class coupling*nya tinggi. Akan tetapi, secara keseluruhan desain aplikasi mencapai *class coupling* rendah yang ditunjukkan oleh $\texttt{CBO} = 0$ yaitu sebanyak 14 pasang. Sedangkan, nilai $\texttt{CBO} \neq 0$ sebanyak 7. Oleh karena itu, bisa disimpulkan desain aplikasi yang diusulkan memiliki nilai *coupling* yang rendah.

6. KESIMPULAN

Penerapan pola desain pada desain aplikasi cuaca nirkabel telah dilakukan. Berdasarkan analisis kelas melalui skenario *use case*, dipakai 2 pola desain yaitu *Abstract Factory* dan *Mediator*. Hasil penerapan pola desain adalah rancangan struktural aplikasi berupa diagram kelas dan rancangan *behavioural* berupa diagram sekuens. Berdasarkan metrik kohesi dan *coupling* didapatkan desain aplikasi cuaca nirkabel yang berkualitas.

Penelitian lebih lanjut yang menarik untuk dilakukan diantaranya: penemuan atau penerapan pola desain untuk aplikasi *embedded system* dan perumusan metrik kohesi dan *coupling* untuk disain berbasis objek.

Daftar Pustaka

[1] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 5th ed. McGraw-Hill Higher Education, 2001.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pattern : Element of Reusable Object-oriented Software*. Addison Wesley, 1994.

[3] A. Heryana, S. Arif, and L. D. Febriani, "Implementasi qt embedded linux pada sbc alix 3d3 sebagai antarmuka grafis stasiun cuaca nirkabel," in *Prosiding Seminar Nasional Embedded System*, 2012, pp. 63–68.

[4] A. Ampatzoglou and A. Chatzigeorgiou, "Evaluation of object-oriented design patterns in game development," *Information and Software Technology*, pp. 445–454, 2007.

[5] M. J. Pont and M. P. Banner, "Designing embedded systems using patterns: A case study," *Journal of Systems and Software*, pp. 201–213, 2004.

[6] A. Ampatzoglou, G. Frantzeskoua, and I. Stamelos, "A methodology to assess the impact of design patterns on software quality," *Information and Software Technology*, pp. 331–346, 2012.

[7] E. G. Nilsson, "Design patterns for user interface for mobile applications," *Advances in Engineering Software*, pp. 1318–1328, 2009.

[8] S. Ahmed and G. Ashraf, "Model-based user interface engineering with design patterns," *Journal of Systems and Software*, pp. 1408–1422, 2007.

[9] M. K. Dalheimer, *Programming wit Qt, 2nd Edition. Writing Portable GUI application on Unix and Win32*. O' Reilly Media, 2002.

[10] P. A. Ezust, *An Introduction to Design Pattern in C++ with Qt 4*. Prentice Hall, 2006.

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transaction on Software Engineering*, pp. 476–493, 1994.