Desain Perangkat Lunak : Konsep dan Tantangannya

Panji Wisnu Wirawan, Satriyo Adhy

Departemen Ilmu Komputer/Informatika Universitas Diponegoro
maspanji@undip.ac.id, satriyo@live.undip.ac.id

Abstract. Desain perangkat lunak merupakan tahapan pengembangan perangkat lunak yang hasilnya akan digunakan oleh pengembang perangkat lunak untuk membuat program. Dalam tulisan ini, disajikan berbagai konsep penting mengenai desain perangkat lunak, termasuk proses desain itu sendiri. Tulisan ini diakhiri dengan mengkaji berbagai tantangan dalam desain perangkat lunak, dalam hal bagaimana merancang perangkat lunak secara efektif dan perancangan perangkat lunak untuk *embedded system*. Dari kajian tersebut diharapkan memicu riset-riset dalam desain perangkat lunak.

1 Pendahuluan

Perangkat lunak umumnya merupakan usaha untuk menyelesaikan permasalahan pada dunia nyata menggunakan komputer. Pengembangan perangkat lunak (*software development*) melalui serangkaian tahapan dimana masing-masing tahapan menghasilkan artifak atau luaran tertentu. Dimulai dari pemahaman masalah (requirement elicitation), analisis, desain, implementasi, dan diakhiri dengan pengujian. Selanjutnya, perangkat lunak ditempatkan (*deploy*) pada pelanggan dan dilakukan pemiliharaan terhadapnya.

Luaran dari tahap requirement elicitation menjadi masukan pada tahapan analisis. Bagian ini menjadi penting karena merupakan tahapan pemahaman terhadap domain masalah yang akan diselesaikan oleh perangkat lunak. Selanjutnya, luaran dari tahapan analisis digunakan pada tahap desain. Proses inipun tidak kalah pentingnya karena hasilnya digunakan sebagai acuan untuk membuat implementasi perangkat lunak.

Tahapan desain menerjemahkan kebutuhan perangkat lunak ke dalam model [1] yang dapat dipahami oleh pengembang perangkat lunak. Beberapa hal harus diperhatikan oleh desainer perangkat lunak supaya perangkat lunak yang dikembangkan dapat fleksibel dan komponen-komponen di dalamnya dapat digunakan ulang (*reuseable*). Pada tulisan ini, dipaparkan beberapa hal terkait dengan desain perangkat lunak yaitu prinsip-prinsip penting, proses desain, dan artifak-artifak yang dihasilkan dalam tahapan desain. Tulisan ini ditutup dengan beberapa tantangan-tantangan dalam desain perangkat lunak

2 Prinsip-Prinsip Desain

Desain perangkat lunak, baik desain konvensional maupun berorientasi objek, dilakukan dengan mengacu pada prinsip-prinsip atau pedoman-pedoman tertentu untuk mempermudah proses desain itu sendiri dan untuk menghasilkan desain berkualitas tinggi. Prinsip-prinsip tersebut merupakan prinsip-prinsip yang umum yang dapat diterapkan pada proyek apapun[1]. Pada bagian ini akan dipaparkan beberapa prinsip umum desain perangkat lunak. Khusus desain berorientasi objek terdapat tambahan beberapa prinsip lain yang akan dipaparkan setelahnya.

2.1 Prinsip-Prinsip Umum

Prinsip-prinsip desain yang umum dapat menjadi pedoman bagi para perancang perangkat lunak dalam membentuk model desain. Pada *Software Engineering Body of Knowledge* (SWEBOK) prinsip perancangan perangkat lunak adalah *abstraction*, *coupling & cohesion*, *decomposition & modularisation*, *encapsulation*, *separation of interface and implementation*, *sufficiency, completeness, & primitiveness* serta *separation of concern* [2].

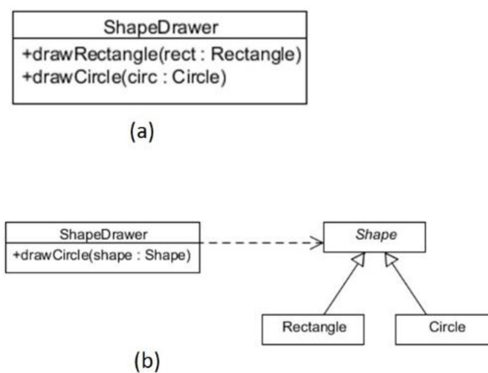
1. *Abstraction*
   Abstraction (abstraksi) terkait dengan bagaimana berfokus dalam memandang objek dan mengambil hal yang penting dari objek tersebut. Tiga macam abstraksi yang dikenal adalah : abstraksi prosedur, data, dan kontrol (iterasi).
2. *Coupling & Cohesion*
   Coupling merupakan ketergantungan antar modul sedangkan *cohesion* merupakan keterikatan antara elemen penyusun modul.
3. *Decompositon & modularization*
   Prinsip ini menekankan pada penguraian (*decompose*) perangkat lunak yang 'besar' menjadi modul-modul atau elemen-elemen dimana masing-masing elemen memiliki fungsi dan tanggung jawab masing-masing.
4. *Encapsulation*
   Prinsip *encapsulation* berarti detail dari sebuah abstraksi tidak diketahui atau tidak dapat diakses oleh entitas yang lain di luarnya.
5. *Separation of interface and implementation*
   Dari sisi komponen perangkat lunak, prinsip ini berarti akses kepada sebuah komponen dari komponen yang lain melalui *public interface* yang telah didefinisikan pada komponen yang akan diakses tersebut.
6. *Sufficiency, completeness & primitiveness*
   Sufficiency dan *completeness* berarti abstraksi yang dilakukan telah menangkap semua karakteristik yang diperlukan sedangkan *primitiveness* artinya desain dapat diimplementasikan.
7. *Separation of concern*
   Prinsip ini terkait dengan arsitektur, dimana terdapat beberapa *architectural view* yang memudahkan stakeholder dalam mengelola kompleksitas perangkat lunak.

2.2 Prinsip Desain Berorientasi Objek

Desain berorientasi objek memiliki prinsip umum ditambah dengan prinsip lain yang dikembangkan berdasarkan konsep berorientasi objek seperti pewarisan (*inheritance*) dan polimorfisme. Beberapa prinsip yang akan dibahas adalah *Single Responsibility Principle*, *Open Closed Principle*, *Liskov Subtitution Principle*, *Dependency Inversion Principle* dan *Interface Segregation Principle* [3].

Single Responsibility Principle (SRP), merupakan prinsip dimana sebuah kelas (*class*) hanya memiliki satu alasan untuk berubah. Artinya bahwa sebuah kelas hanya memiliki sebuah tanggung jawab tertentu. Contoh sebuah tanggung jawab adalah sebuah *method /* perilaku. Ketika sebuah kelas memiliki banyak *method* dengan perilaku yang bermacam-macam, maka akan tercipta ketergantungan antar *method* di dalam kelas tersebut. Akibatnya, perubahan di satu *method*, akan mempengaruhi *method* yang lain.

Open Closed Principle (OCP), menekankan bahwa pengembangan/perluasan yang dilakukan pada entitas perangkat lunak seperti kelas dan modul semestinya melalui *extension* bukan melalui penyuntingan kode program. Sebagai contoh *extension* adalah menggunakan konsep pewarisan ataupun polimorfisme. Gambar 1 adalah salah satu contoh pelanggaran konsep OCP (gambar 1.a) dan yang memenuhi konsep OCP (gambar 1.b).



Gambar 1. Sebuah contoh untuk desain kelas untuk menggambar *Rectangle* dan *Circle*. Gambar (a) merupakan sebuah kelas yang melanggar prinsip OCP karena dalam kelas tersebut ketika akan ada perubahan atau mungkin penambahan sebuah method, yang dilakukan adalah menyunting kelas ShapeDrawer. Hal tersebut tidak diperkenankan karena dapat mempengaruhi method-method yang lain. Sedangkan gambar (b) merupakan desain yang memenuhi prinsip OCP karena dalam desain tersebut, perubahan yang dilakukan pada sebuah kelas terisolasi pada kelas tersebut, sedangkan untuk menambahkan satu fungsi gambar yang lain, dapat dilakukan dengan membuat kelas baru dan menurunkan kelas abstrak Shape.

Liskov Subtitution Principle (LSP), adalah prinsip dimana supertipe (*supertype*) dapat mensubstitusi subtipe (*subtipe*). Jika diterapkan pada pewarisan, maka *superclass*

harus dapat mensubstitusi *subclass*-nya. Artinya di sini bahwa hubungan antara *subclass* dan *superclass* harus memenuhi kaidah “IS-A”

Dependency Inversion Principle (DIP) merupakan prinsip yang menekankan pada penggunaan abstraksi. Artinya, ketergantungan yang dibangun pada hubungan antar kelas semestinya kepada kelas abstrak atau *interface*, bukan pada kelas konkrit (*concrete class*).

Prinsip yang terakhir, *Interface Segregation Principle* (ISP), adalah sebuah prinsip yang memandu pembuatan *interface*. Prinsip ini tidak menyarankan pembuatan *interface* yang terlalu banyak *method* yang tidak perlu. Lebih baik, *method-method* yang tidak berkaitan dipisahkan dalam *interface* yang lain.

3 Proses Desain

Desain bertujuan untuk membentuk sebuah model yang siap untuk diimplementasikan ke dalam program. Dalam membentuk model desain, terdapat serangkaian proses yang perlu dilakukan dengan tetap berpegang pada prinsip-prinsip desain. Proses desain menurut SWEBOK terdiri atas dua aktivitas yaitu software *architectural design* dan *software detailed design* [2]. Semua kegiatan desain perlu untuk didokumentasikan dan proses dokumentasi perlu manajemen yang baik[1].

Dalam desain berorientasi objek diuraikan beberapa kegiatan yang lain pada tahapan desain [4]. Kegiatan tersebut adalah :

- Mendefinisikan tujuan desain.
- Mendefinisikan subsistem.
- Pemetaan subsistem ke dalam *platform* yang digunakan
- Pengelolaan *persistent data*.
- Mendefinisikan kendali akses.
- Mendefinisikan kendali aliran (*control flow*).
- Mendefinisikan *boundary condition*.

Kegiatan-kegiatan tersebut merupakan uraian dari dua kegiatan utama pada tahap desain. Berikutnya, akan diurakan dua kegiatan utama dari tahapan desain tersebut.

3.1 Desain Arsitektur

Desain arsitektur merupakan desain makro / struktur yang mencerminkan kualitas serta fungsi dari perangkat lunak. Aktivitas pembentukan arsitektur merupakan aktivitas dekomposisi, yaitu membagi perangkat lunak menjadi elemen-elemen[2]. Terdapat panduan dalam aktivitas pembentukan desain arsitektur [5] :

- Memikirkan apa (*what*) yang akan dilakukan oleh sistem menjadi pertimbangan utama daripada memikirkan bagaimana (*how*) sistem melakukan sesuatu.
- Desain abstrak (*interface*, kelas abstrak atau abstrak data type) dipertimbangkan lebih dahulu daripada desain yang konkrit (*concrete class*).

- Kebutuhan non fungsional digunakan sebagai acuan dalam awal proses desain.
- Pemakaian ulang sebanyak mungkin elemen/komponen menjadi pertimbangan dalam menyusun desain.
- Desain elemen arsitektur dilakukan dengan mengutamakan *high cohesion* dan *loose coupling*.
- Desain disusun tidak dalam satu proses namun dalam beberapa proses yang berulang.
- Desain arsitektur yang telah disusun tidak ambigu dan terlalu detail (*over detailed*)

Arsitektur menggambarkan elemen-elemen penyusun perangkat lunak berikut hubungan antar elemen tersebut atau bagaimana antar elemen saling berkomunikasi. Hasil dari kegiatan arsitektur dapat digunakan sebagai alat evaluasi oleh para pemangku kepentingan (*stakeholders*) terutama yang terkait dengan kebutuhan nonfungsional.

Setidaknya terdapat “4+1” sudut pandang (*view model*) dalam arsitektur perangkat lunak yaitu *logical*, *process*, *physical*, *development* serta *secenario* / *use case* [6]. Masing-masing view tersebut dapat dimanfaatkan oleh perancang perangkat lunak untuk membuat deskripsi mengenai arsitektur yang dipilih, dari masing-masing sudut pandang.

- *Use case* / *scenario* : merupakan bentuk dari keseluruhan fungsi perangkat lunak dan digunakan dasar acuan dari pembentukan *logical*, *process*, *physical* maupun *development view*. Selain sebagai dasar acuan, *scenario*, menjadi alat untuk validasi setelah keseluruhan *view* selesai didokumentasikan.
- *Logical view* : menggambarkan kebutuhan fungsional, yaitu kebutuhan yang semestinya diberikan oleh sistem kepada *end user*. Penggambaran berupa objek dan kelas yang menggunakan enkapsulasi maupun pewarisan.
- *Process view* : memungkinkan desainer untuk menggambarkan beberapa kebutuhan non fungsional seperti *performance* dan *availability*. Dalam view ini juga terdapat definisi thread of control yang mengendalikan objek maupun kelas dalam *logical view*.
- *Physical view* : mengilustrasikan kebutuhan non fungsional seperti *scalability* dan *reliability*. Pada *physical view*, hal-hal yang telah didefinisikan dalam *logical*, *process* dan *development view* dipetakan dalam node-node (mesin-mesin fisik ataupun *platform*).
- *Development view* : berfokus pada organisasi perangkat lunak dalam modul-modul atau sub-sistem, yang akan diterapkan pada perangkat lunak yang dikembangkan.

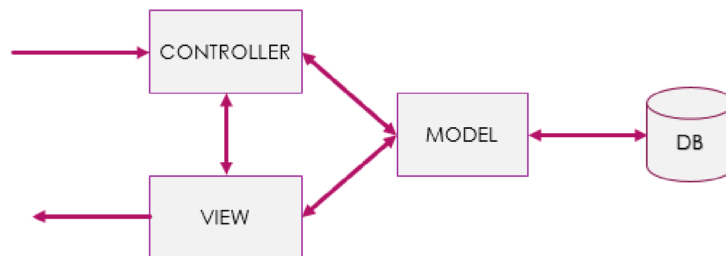
Penyusunan perangkat lunak dalam modul-modul beserta konektornya akan membentuk struktur perangkat lunak. Dalam proyek perangkat lunak, akan dapat ditemukan struktur-struktur yang mirip atau hampir sama dan dapat digunakan ulang untuk proyek lain. Strukur-struktur tersebut dikenal dengan *architectural style* [5].

Terdapat banyak kelompok *architectural style* yang telah didefinisikan. Beberapa kelompok tersebut, beserta *architectural style*-nya adalah ditunjukkan pada tabel 1.

Masing-masing architectural style memiliki elemen-elemen tertentu dengan pola-pola konektor dan aturan komunikasi antar elemen tertentu. Sebagai contoh, Model-View-Controller (MVC) memeiliki elemen-elemen seperti model, view dan controller dengan pola komunikasi yang ditunjukkan pada Gambar 2.

Tabel 1. Beberapa kelompok arsitektur dan architectural style[5] .

Kelompok Arsitektur	*Architectural Style*
Data Flow	Pipe & Filter, Process Control
Data-Centered	Repository, Blackboard
Hierarchical	Layered, Master-Subroutine
Interaction Oriented	Model-View-Controller (MVC)
Distributed	Client-Server, Multi-tiers, Broker
Implicit Asynchronous Communication	Buffered Message-Based, Non-buffered event-based implicit invocation



Gambar 2. *Model-View-Controller architectural style*, dimana anak panah menunjukkan arah komunikasi dari masing-masing komponen.

3.2 Desain Mendetail

Tahapan desain secara mendetail (*detailed design*) dilakukan setelah tahapan desain arsitektur telah dilakukan. Pada tahap ini, setiap komponen didefinisikan detailnya sampai pada tahap yang bisa diimplementasikan ke dalam program [2]. Sebagai contoh, pada arsitektur MVC, tahapan *detailed design* mendefinisikan kelas ataupun *interface* yang terlibat (*internal structure*) untuk membentuk fungsi elemen controller, view, dan model. Kelas atau *interface* tersebut nantinya dapat diimplementasikan ke dalam program yang membentuk masing-masing fungsi elemen.

Kegiatan pada desain mendetail dapat dibagi menjadi dua yaitu desain *interface* dan *komponen* [1]. Desain *interface* menghasilkan berbagai cara akses untuk berkomunikasi dengan komponen, baik di dalam komponen sendiri ataupun akses untuk komponen lain di luar komponen tersebut. Di sisi lain, desain komponen menghasilkan detail dari masing-masing komponen/elemen serta proses perbaikan (*refinement*) dari komponen-komponen yang dihasilkan dari proses sebelumnya.

Pada desain berorientasi objek, masing-masing elemen pada arsitektur tersusun dari kelas-kelas maupun *interface*. Sama halnya seperti elemen penyusun arsitektur, elemen-elemen yang membentuk struktur internal elemen ini pun memiliki pola yang berulang yang dinamakan *design pattern* [7] . Setidaknya terdapat 3 kelompok pola dalam *design pattern*. berorientasi objek yaitu *structural*, *behavioral* dan *creational pattern*. Masing-masing kelompok dengan *design pattern* yang sesuai ditunjukkan pada tabel 2.

Tabel 2. Beberapa kelompok design pattern dan pattern yang bersesuaian .

Kelompok *design pattern*	*Design Pattern*
Creational	Prototype, Singleton, Factory, Builder
Behavioral	Observer, Strategy, State, Memento, Mediator
Structural	Adapter, Bridge, Decorator, Facade

4 Model Desain

Hasil akhir dari proses desain adalah model desain. Pada model desain berorientasi objek, model desain menggunakan diagram Unified Modelling Language (UML). Elemen-elemen model desain dapat dibagi berdasarkan dimensi prosesnya, yaitu *architecture*, *interface*, *component level*, dan *deployment level* [8]. Tabel 3 menunjukkan model desain berorientasi objek berdasarkan dimensi prosesnya yang sebagian besar merupakan diagram UML.

Tabel 3. Model desain dari berbagai sudut pandang proses .

Dimensi Proses	Model desain
Architecture	Class realization, Subsystems, Collaboration Diagram
Interface	Technical interface design, navigation design, GUI design
Component level	Component diagram, design classes, activity diagram, sequence diagram.
Deployment level	Deployment diagram

5 Tantangan Desain Perangkat Lunak

Pengembangan perangkat lunak memiliki sejumlah tantangan yang dapat dilihat dari berbagai aspek sudut pandang. Berikut contoh sejumlah aspek dilihat dari dua contoh kelompok, yaitu dari kelompok desain yang efektif dan *embedded system*.

5.1 Desain yang Efektif

Expertise in design : Perkembangan studi empiris dan formal berkaitan dengan perilaku desainer telah meningkat. Studi keahlian desainer telah dilihat mulai dari desainer pemula dibandingkan dengan desainer ahli, perilaku desainer ahli dan desainer yang sangat ahli. Pada sisi lain, keahlian mendesain memiliki beberapa aspek yang berbeda dari keahlian pada bidang yang lain [9]. Perilaku desainer ini menentukan aktivitas dan kecepatan pekerjaan yang dilakukan. Proses menstrukturisasi dan memformulasikan masalah dapat diidentifikasi sebagai hal yang penting berkaitan dengan keahlian mendesain, konsep "*problem framing*" dapat menjadi kata kunci untuk mendeskripsikan hal ini. Kesuksesan, pengalaman, dan khususnya seorang desainer ahli akan lebih pro-aktif didalam "*problem framing*", secara aktif melihat masalah dan mengarahkan untuk membuat solusinya. Merupakan sebuah tantangan perilaku mendesain untuk selalu pro-aktif melihat masalah kemudian melakukan "*problem framing*" dan membuat solusinya, pengalaman proyek akan membangun karakteristik seseorang dalam hal ini.

Representing structure in a software system design : struktur merupakan alat kunci utama didalam desain dari sebuah artifak yang komplek. Produk esensial dari seorang software designer tidak hanya software itu sendiri namun perilaku yang terjadi pada "masalah dunia nyata" diluar komputer dan efek yang disajikan kepada pengguna nantinya. Properti dari "masalah dunia nyata", pemahaman kapasitas pengguna, dan ekploitasi fungsi komplek sistem merupakan subjek vital untuk menjadi perhatian seorang desainer[10].

Effective software design : desain software merupakan sebuah proses kognitif yang komplek, dimana pembuatan keputusan memiliki peran yang sangat penting, akan tetapi pemahaman tentang bagaimana pembuatan keputusan tersebut sangat terbatas. Beberapa faktor yang mempengaruhi efektifitas desain software yaitu : rencana desain (*design planning*), *design context switching, problem-solution co-evolution* dan *teknik application of reasoning* [11]. Pada sisi lain, sebuah model deskriptif tentang pembuatan keputusan oleh seorang desainer software telah dikembangkan [12]. Dapatkah model deskriptif tersebut menjawab bagaimana seharusnya sebuah keputusan dibuat pada saat desain perangkat lunak dilakukan?

Ideas, Subjects, Cycles : di dalam proses desain perangkat lunak, sering dapat diamati bahwa seorang desainer mempertimbangkan dua buah subjek dalam satu waktu serta memutar putar subjek dalam proses implementasinya, sesinya juga dapat dikarakteristikkan memiliki perulangan yang banyak, desainer juga secara reguler menyatakan ulang ide yang telah ada dalam konteks yang lain[13]. Pada bagian ini

menyatakan bagaimana sebuah ide itu dapat diubah karena dilihat dari sudut pandang yang lain, sedangkan subject dapat dilihat secara bersamaan dan saling dipertukarkan prioritas pengimplementasiannya, dan baik ide maupun subject dapat memiliki iterasi yang sangat banyak. Tantangannya yaitu bagaimana seorang desainer software dapat mengelola ketiga hal tersebut dengan cermat, tepat, dan cepat.

Architecture design rationale : banyak yang telah mengklaim berkaitan dengan konsekuensi yang harus ditanggung jika tidak mendokumentasikan desain pemikiran (*design rationale*). Persepsi umumnya adalah desainer dan arsitek biasanya tidak memahami sepenuhnya peran kritis dari penggunaan dan pendokumentasian *design rationale*. Penelitian kedepan dibutuhkan pengembangan metodologi dan perkakas yang cocok untuk mendukung pendokumentasian dan penggunaan *design rationale* [14].

General model of software architecture design : telah dilakukan perbandingan dari 5 metode desain arsitektur perangkat lunak industri dan melakukan ekstraksi berdasarkan kesamaan sebuah model umum pendekatan desain arsitektur software. Berdasarkan pola ideal yang ada dapat disajikan grid evaluasi yang dapat digunakan untuk perbandingan metode kedepannya [15]. Perbandingan metode masih perlu divalidasi lebih lanjut untuk memastikan model umum tersebut betul-betul dapat digunakan.

5.2 Embedded System

Tools selection in embedded systems : *Model based system engineering* (MBSE) merupakan sebuah pendekatan sistematik dari modeling yang secara teratur digunakan untuk mendukung aktifitas pembangunan system mulai dari *requirement specification, design, verification* dan *validation*. Pada sisi lain, merupakan pekerjaan yang sulit untuk mengkostumisasi MBSE untuk pengembangan *embedded system* dikarenakan perbedaan aspek perilaku perangkat lunaknya. Sebuah tantangan untuk dapat memilih perkakas yang cocok dalam melakukan aktifitas MBSE. Investigasi komprehensif berkaitan dengan pemilihan perkakas ini telah dilakukan dengan harapan dapat memfasilitasi peneliti, praktisi, dan pengembang untuk dapat memilih perkakas yang cocok sesuai dengan kebutuhannya masing-masing [16].

A software integration approach : *Embedded system* semakin menyajikan isu kompleks berkaitan dengan *hardware-software* (HW-SW) *co-design*. Berbagai macam hal yang menjadi kajian dan isu pada *embedded system* menjadi sebuah tantangan kedepan yaitu : *resource sharing, co-design solution, processing capability, power, communication bandwidth, precedence realtions, real-time deadlines, space,* dan *cost* [17].

6 Kesimpulan

Desain perangkat lunak merupakan salah satu tahapan pengembangan perangkat lunak yang menjadi jembatan dari analisis dan implementasi program. Detail teknis dihasilkan dari desain perangkat lunak sehingga pengembang perangkat lunak mudah mengimplementasikan atau membuat perangkat lunak. Dalam beberapa hal desain perangkat lunak memiliki tantangan tersendiri, sebagai contoh dalam desain yang efektif dan dasain perangkat lunak *embedded system*.

Referensi

[1] C. . Otero, *Software Engineering Design : Theory and Practice*. CRC Press, 2012.

[2] P. Bourque and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014.

[3] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[4] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java*, 3rd ed. Pearson, 2009.

[5] K. Qian, X. Fu, L. Tao, and C. Xu, *Software Architecture And Design Illuminated*. Jones & Bartlett Learning, 2009.

[6] P. B. Kruchten, “The 4+1 View Model of architecture,” *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, 1995.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[8] R. S. Pressman, *Software Engineering A Practitioner’s Approach*, 7th ed. McGraw-Hill, 2010.

[9] N. Cross, “Expertise in design: an overview,” *Des. Stud.*, vol. 25, no. 5, pp. 427–441, Sep. 2004.

[10] M. Jackson, “Representing structure in a software system design,” *Des. Stud.*, vol. 31, no. 6, pp. 545–566, Nov. 2010.

[11] A. Tang, A. Aleti, J. Burge, and H. van Vliet, “What makes software design effective?,” *Des. Stud.*, vol. 31, no. 6, pp. 614–640, Nov. 2010.

[12] H. Christiaans and R. A. Almendra, “Accessing decision-making in software design,” *Des. Stud.*, vol. 31, no. 6, pp. 641–662, Nov. 2010.

[13] A. Baker and A. van der Hoek, “Ideas, subjects, and cycles as lenses for understanding the software design process,” *Des. Stud.*, vol. 31, no. 6, pp. 590–613, Nov. 2010.

[14] A. Tang, M. A. Babar, I. Gorton, and J. Han, “A survey of architecture design rationale,” *J. Syst. Softw.*, vol. 79, no. 12, pp. 1792–1804, Dec. 2006.

[15] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America, “A general model of software architecture design derived from five industrial approaches,” *J. Syst. Softw.*, vol. 80, no. 1, pp. 106–126, Jan. 2007.

[16] M. Rashid, M. W. Anwar, and A. M. Khan, “Toward the tools selection in model based system engineering for embedded systems—A systematic literature review,” *J. Syst. Softw.*, vol. 106, pp. 150–163, Aug. 2015.
[17] N. Suri, A. Jhumka, M. Hiller, A. Pataricza, S. Islam, and C. Sârbu, “A software integration approach for designing and assessing dependable embedded systems,” *J. Syst. Softw.*, vol. 83, no. 10, pp. 1780–1800, Oct. 2010.