KONSEP DASAR, KONTEKS, DAN PRINSIP PERANCANGAN PERANGKAT LUNAK

Mata Kuliah: Software Engineering

DOSEN: Yudhi Fajar Saputra, S.Kom., M.Sc

Pertemuan ke-6

Topik Bahasan ke-14

SEMESTER : 3/ TA. 2024-2025

KODE MK/SKS: MKP001/3 SKS

PRODI INFORMATIKA/ILMU KOMPUTER

UNIVERSITAS WIDYA GAMA MAHAKAM SAMARINDA

Nama Mata Kuliah	: Software Engineeing/Rekayasa Perangkat Lunak
Kode Mata Kuliah/SKS	: **MKP _____/3 SKS**
Dosen	: **Yudhi Fajar Saputra,**
Semester	: **3/ 2024**
Hari Pertemuan / Jam	: -
Tempat Pertemuan	: **Ruang Kelas A.06**

Konsep dasar perancangan perangkat lunak mencakup berbagai prinsip dan metode yang digunakan untuk mengembangkan perangkat lunak yang berkualitas, efisien, dan dapat diandalkan, sehingga akan menjadi fondasi yang kuat untuk membangun perangkat lunak yang berkualitas, sementara konteks perancangan perangkat lunak merujuk pada environment, batasan, dan faktor-faktor yang mempengaruhi proses pembuatan perangkat lunak dimana aspek-aspek tersebut bisa mempengaruhi bagaimana perangkat lunak dirancang dan dikembangkan, kemudian Prinsip-prinsip perancangan perangkat lunak adalah pedoman umum yang harus dipatuhi dalam proses pengembangan perangkat lunak. Prinsip-prinsip ini bertujuan untuk menghasilkan perangkat lunak yang berkualitas, mudah dipelihara, dan dapat diandalkan. Berikut adalah penjelasan lebih rinci mengenai konsep dasar, konteks, proses, dan prinsip perancangan perangkat lunak

1. KONSEP DASAR PERANCANGAN PERANGKAT LUNAK

Konsep dasar perancangan perangkat lunak berfungsi sebagai panduan untuk menciptakan perangkat lunak yang berkualitas, scalable, dan mudah dipelihara. Dengan memahami dan menerapkan konsep-konsep ini pengembangan dapat merancang sistem perangkat lunak yang efektif dan efisien, selain itu pengembang juga dapat merancang sistem yang tidak hanya memenuhi kebutuhan pengguna saat ini tetapi juga dapat berkembang dan beradaptasi dengan perubahan di masa depan. Berikut adalah beberapa konsep dasar yang penting dalam perancangan perangkat lunak:

a. Modularitas: Modularitas adalah prinsip membagi perangkat lunak menjadi bagian-bagian kecil yang independen, bagian-bagian kecil tersebut dinamakan dengan modul. Setiap modul memiliki tugas terhadap bagian tertentu dari sistem sehingga dapat dikembangkan, diuji, serta dipelihara secara terpisah. Erich Gamma et al, menjelaskan bahwa Modularitas merupakan langkah pembagian perangkat lunak menjadi bagian-bagian kecil yang lebih mudah dikelola dan dipahami. Tiap modul memiliki tugas spesifik, yang sangat berguna bagi pengembang untuk memodifikasi bagian tertentu tanpa mempengaruhi keseluruhan sistem[1]. Dengan modularitas, pengembang dapat mengelola

kompleksitas sistem dengan lebih simpel sehingga memudahkan meningkatkan kemudahan dalam memahami ketika proses pengembangan, pengujian, pemeliharaan, dan hal tersebut sangat berguna dalam pembagian tugas bagi tim pengembangan.

b. Abstraksi. Abstraksi adalah proses penyederhanaan yang melibatkan penghapusan detail-detail teknis yang tidak relevan untuk fokus pada aspek penting dari sebuah sistem. Dalam desain perangkat lunak, abstraksi memungkinkan pengembang untuk memahami dan bekerja pada bagian tertentu dari sistem tanpa harus memahami semua detail di baliknya. Abstraksi melibatkan penyederhanaan masalah kompleks dengan menyembunyikan detail-detail implementasi yang tidak perlu bagi pengguna akhir. Ini memungkinkan pengembang untuk fokus pada level yang lebih tinggi dari desain tanpa terjebak dalam detail teknis [2]. Dalam desain perangkat lunak, abstraksi sangat berguna bagi pengembang untuk memahami dan bekerja pada bagian tertentu dari sistem tanpa harus memahami semua detail di baliknya sehingga dapat mengurangi kompleksitas dan tentunya pengembang fokus pada level yang lebih tinggi dari desain, seperti antarmuka dan interaksi antar modul.

c. Encapsulation. Encapsulation adalah suatu proses menyembunyikan atau mengkapsulkan data – data kelas ke dalam suatu unit. Artinya konsep di mana data dan metode yang terkait dikelompokkan dalam satu unit (misalnya, dalam kelas dalam OOP). Pengkapsulan berguna bagi pengembang untuk menyembunyikan detail implementasi dalam modul atau kelas dengan menyediakan antarmuka publik yang bersih dan sederhana [3], maka dari itu pengkapsulan melindungi data dengan membatasi akses langsung dan hanya memperbolehkan akses melalui antarmuka yang ditentukan.

d. Inheritance dan Hierarki: Inheritance atau Pewarisan/Penurunan adalah konsep pemrograman dimana sebuah class dapat menurunkan properti, metode, dan atribut yang dimilikinya kepada class lain . Sedangkan Hierarki adalah struktur di mana kelas-kelas diorganisasikan dalam bentuk pohon, dengan kelas induk (superclass) di atas dan kelas turunan (subclass) di bawah, maka dari itu dalam desain berorientasi objek, hierarki digunakan untuk mengatur kelas dalam struktur yang bisa memberikan pewarisan sifat dan perilaku dari kelas induk ke kelas turunan [4]. Dengan begitu Inheritance sangat berguna dalam penggunaan kembali kode yang sudah ada.

e. Koherensi (Cohesion)

Koherensi merujuk pada tingkat keterpaduan dan fokus elemen-elemen dalam satu modul atau kelas, yang artinya seberapa terkait dan terfokusnya fungsi-fungsi dalam satu modul. Modul dengan koherensi tinggi memiliki elemen-

elemen yang saling bekerja sama untuk mencapai satu tujuan tertentu [5].

f. Kopling (Coupling)

Kopling adalah tingkat saling ketergantungan satu modul terhadap modul lain. Dalam pengembangan aplikasi atau software harus sebisa mungkin memiliki Kopling rendah (loose coupling) karena jika ada perubahan pada satu modul, maka tidak akan secara signifikan mempengaruhi modul-modul lain [2], maka dari itu semakin rendah Module Coupling, semakin baik fungsinya.

g. Antarmuka (Interface)

Antarmuka adalah bagian dari perangkat lunak yang berfungsi untuk komunikasi antara modul-modul yang berbeda, atau antara perangkat lunak dan pengguna, dengan kata lain bagaimana modul-modul perangkat lunak berinteraksi [4].

h. Reusability (Penggunaan Ulang)

Reusability adalah konsep di mana komponen perangkat lunak dirancang untuk digunakan kembali dalam berbagai proyek, termasuk desain modular dan abstraksi. Ini mengurangi biaya dan mempercepat waktu pengembangan karena mengurangi duplikasi kerja, serta meningkatkan kualitas karena memanfaatkan komponen yang telah teruji [6].

i. Refactoring

Refactoring adalah proses perbaikan struktur internal kode tanpa mengubah fungsionalitas eksternalnya. Ini dilakukan untuk meningkatkan kualitas kode, seperti mengurangi kompleksitas, meningkatkan koherensi, dan menurunkan kopling [7]. Refactoring juga membuat kode lebih bersih, lebih mudah dipahami, dan lebih mudah dipelihara

j. Desain untuk Perubahan (Design for Change)

Konsep ini mengacu pada perancangan perangkat lunak dengan memperkirakan kemungkinan perubahan di masa yang akan datang sehingga lebih adaptable. Untuk itu penggunaan teknik yang mempermudah modifikasi dan pengembangan sistem seiring waktu sangat diperlukan, teknik-teknik ini mencakup seperti penggunaan pola desain (design patterns) dan pengkodean yang modular dan fleksibel [7]. Maka dari perangkat lunak harus memiliki dasain for change yang dapat berkembang dan menyesuaikan diri dengan perubahan kebutuhan atau lingkungan operasi tanpa memerlukan perombakan besar-besaran.

2. KONTEKS PERANCANGAN PERANGKAT LUNAK

Konteks perancangan perangkat lunak adalah aspek-aspek yang memengaruhi bagaimana perangkat lunak dirancang dan dikembangkan, Konteks perancangan

perangkat lunak mencakup tentang berikut ini: kebutuhan pengguna, tujuan bisnis, environment, dan batasan teknis [3,4]. Selain itu, aspek-aspek seperti regulasi, keamanan, User Experience, dan batasan waktu serta anggaran juga memainkan peran penting dalam membentuk desain perangkat lunak [8]. Dengan memahami konteks ini, pengembang dapat membuat keputusan yang tepat untuk menghasilkan perangkat lunak yang memenuhi kebutuhan, berfungsi dengan baik, dan dapat beradaptasi dengan perubahan di masa depan.

a. Kebutuhan Pengguna: Setiap perancangan perangkat lunak harus dimulai dari pemahaman tentang kebutuhan pengguna. Pengguna dapat berupa individu, kelompok, atau organisasi yang akan menggunakan perangkat lunak tersebut.
b. Tujuan bisnis: Perangkat lunak sering kali dirancang untuk mendukung tujuan bisnis tertentu, seperti meningkatkan efisiensi operasional, mengurangi biaya, atau membuka pasar baru
c. Environment: environment mencakup platform perangkat keras dan perangkat lunak, sistem operasi, jaringan, dan perangkat lain yang akan digunakan untuk merancang perangkat lunak.
d. Batasan Teknis: batasan teknis mencakup batasan yang terkait dengan teknologi yang digunakan, seperti performa, penyimpanan, bandwidth, dan kompatibilitas
e. Regulasi: ada regulasi dan standar tertentu atau aturan yang harus dipatuhi dalam merancangan perangkat lunak, terutama dalam industri yang diatur dengan ketat seperti keuangan, kesehatan, dan pemerintahan
f. Skalabilitas dan Ekstensibilitas: Skalabilitas merujuk pada kemampuan perangkat lunak yang dirancang agar dapat bisa ditingkatkan performa dimasa depan, sedangkan ekstensibilitas adalah kemampuan untuk menambah fitur atau memperluas fungsionalitas di masa depan
g. Keamanan : perancangan perangkat lunak harus mencakup mekanisme untuk melindungi terhadap ancaman keamanan, seperti enkripsi, kontrol akses, dan deteksi serta respons terhadap serangan.
h. User Experience: Perancangan perangkat lunak harus memiliki antarmuka yang intuitif dan responsif, serta memastikan bahwa perangkat lunak memenuhi kebutuhan dan harapan pengguna dengan cara yang efisien dan menyenangkan.
i. Batasan waktu dan anggaran: Perancangan perangkat lunak harus memprioritaskan fitur dan fungsionalitas yang paling penting, serta mencari solusi yang efisien dan efektif untuk memaksimalkan hasil dalam batasan yang waktu dan anggaran yang telah ditetapkan.

3. PRINSIP PERANCANGAN PERANGKAT LUNAK

Perkembangan teknologi dan metodologi pengembangan perangkat lunak yang begitu pesat membuat prinsip-prinsip perancangan juga terus berevolusi, akan tetapi ada beberapa prinsip untuk perancangan perangkat lunak yang tetap digunakan. Berikut adalah beberapa prinsip yang umum masih digunakan:

a. Prinsip Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, dan Dependency Inversion Principle (SOLID)

   Prinsip-prinsip ini memberikan panduan yang lebih detail tentang bagaimana merancang kelas dan modul yang baik, dengan tujuan meningkatkan fleksibilitas, pemeliharaan, dan testabilitas kode. Prinsip ini terdiri empat prinsip yaitu Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, dan Dependency Inversion Principle [3], lebih detail mengenai prinsip dari ke empat prinsip SOLID adalah sebagai berikut:

   1) Single Responsibility Principle (SRP): Prinsip ini menyatakan bahwa setiap kelas atau modul dalam perangkat lunak setidaknya harus memiliki satu tugas. Prinsip ini menitik beratkan terhadap desain yang lebih bersih dan lebih mudah dipelihara yaitu dengan membagi tugas ke dalam unit-unit yang lebih kecil
   2) Open-Closed Principle (OCP): Entitas perangkat lunak (classes, modules, functions, dll) harus terbuka untuk ekstensi tetapi tertutup untuk modifikasi. artinya bahwa perangkat lunak memiliki karakter harus bisa dikembangkan tanpa mengubah kode yang ada, yang membantu menjaga stabilitas dan mencegah kesalahan baru ketika kode dikembangkan.
   3) Liskov Substitution Principle (LSP): Prinsip yang menyatakan bahwa objek dalam program bisa digantikan dengan instansi dari subclass mereka tanpa mengganggu fungsi program. LSP mempunyai prinsip bahwa subclass atau kelas turunan dapat menambah fungsi kelas induk tanpa mengubah fungsi yang diharapkan.
   4) Interface Segregation Principle (ISP): ISP mempunyai prinsip bahwa dalam pengembangan perangkat lunak, antarmuka perangkat lunak dibuat spesifik untuk kebutuhan klien daripada membuat satu antarmuka besar yang memaksakan klien yang tidak mereka butuhkan. Dengan cara ini, perangkat lunak lebih modular dan fleksibel.
   5) Dependency Inversion Principle (DIP): DIP merupakan salah satu prinsip perancangan perangkat lunak yang menerangkan bahwa modul tingkat tinggi tidak boleh bergantung pada modul tingkat rendah. Sebaliknya, kedua jenis modul harus bergantung pada abstraksi (antarmuka atau kelas abstrak). Ini membantu mengurangi ketergantungan yang kuat antar komponen perangkat

lunak, dan sangat berguna agar sistem lebih mudah untuk di modifikasi.

b. Prinsip Clean Code: Konsep ini dipopulerkan oleh Robert C. Martin dalam bukunya yang mana prinsip "Clean Code" memberikan panduan yang komprehensif tentang bagaimana menulis kode [3], yaitu dengan melakukan koding secara bersih, terbaca, dan mudah dipelihara.

c. Prinsip DRY (Don't Repeat Yourself): Prinsip DRY merupakan prinsip perancangan lunak agar menghindari duplikasi dalam kode [9], prinsip ini menekankan untuk memastikan bahwa setiap informasi atau logika memiliki satu representasi tunggal dalam sistem. Sehingga dapat mengurangi kesalahan dan membuat pemeliharaan kode lebih mudah.

d. Prinsip KISS (Keep It Simple, Stupid): KISS merupakan prinsip perancangan perangkat lunak yang menekankan pentingnya menjaga desain perangkat lunak tetap sederhana dan fokus pada tujuan [10]. Dengan prinsip KISS, menekankan bahwa kompleksitas yang tidak perlu harus dihindari untuk membuat perangkat lunak lebih mudah dipahami, dikembangkan, dan dipelihara.

e. Prinsip YAGNI (You Aren't Gonna Need It): Prinsip YAGNI merupakan prinsip perancangan perangkat lunak yang memiliki prinsip untuk tidak menambahkan fitur tambahan yang tidak diperlukan untuk saat ini[11]. Pronsip ini menitik beratkan berfokus pada pengembangan fitur yang benar-benar diperlukan, sehingga dapat mengurangi waktu pengembangan dan kompleksitas perangkat lunak

f. Prinsip Tes-Driven Development (TDD): Prinsip ini merupakan prinsip perancangan perangkat lunak yang menempatkan pengujian sebagai fokus utama sepanjang siklus pengembangan [12], yang artinya prinsip ini didasarkan pada siklus iteratif yang melibatkan pengujian

4. DAFTAR REFERENSI

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1994). Addison-Wesley Professional. ISBN-10: 0201633612.
2. Steve McConnell. (2004). Code Complete: A Practical Handbook of Software Construction, Second Edition. Microsoft Press. ISBN-10: 0735619670
3. Robert C. Martin (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Pearson. ISBN-10: 9780132350884
4. Erich Gamma, et al. (1994). Design Patterns: Elements of Reusable Object-Oriented Software 1st Edition. Addison-Wesley Professional. ISBN-10: 0201633612.
5. Stephen T. Albin. (2003). The Art of Software Architecture. John Wiley &Sons. ISBN-10: 0471228869

6. Ivar Jacobson, Martin Griss, Dr. Patrik Jonsson (1996). Software Reuse: Architecture Process and Organization for Business Success. Addison-Wesley Professional. ISBN-10: 0691129916
7. Martin Fowler (2018). Refactoring: Improving the Design of Existing Code (2nd Edition). Addison-Wesley Professional. ISBN-10: 0134757599
8. Titus Winters, Tom Manshreck, Hyrum Wright. (2020). Software Engineering at Google. O'Reilly Media, Inc. ISBN: 9781492082798
9. Andrew Hunt &David Thomas (1999). The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Professional. ISBN-10: 9780201616224
10. Brian W. Kernighan, P. J. Plauger (1978) . The Elements of Programming Style, 2nd Edition 2nd Edition. McGraw-Hill. ISBN-10: 0070342075
11. Kent Beck, Cynthia Andres (2004). Extreme Programming Explained: Embrace Change, 2nd Edition (The XP Series) 2nd Edition. Addison-Wesley. ISBN-10: 9780321278654
12. Kent Beck (2021). Test-Driven Development: By Example. Addison-Wesley Professional. ISBN: 0321146530
13. IEEE. IEEE Standard 830-1998 - Recommended Practice for Software Requirements Specifications. ISBN:978-0-7381-0448-5. Reterived at 08 August 2024 from https://ieeexplore.ieee.org/document/720574
14. Sommerville, Ian. (2015). Software Engineering 10th Edition. Pearson Education, Inc.. ISBN-13: 978-0-13-703515-1.
15. Karl Wiegers dan Joy Beatty. (2013). Software Requirements (Developer Best Practices) 3rd Edition. Microsoft Press. ISBN-10: 0735679665
16. Roger S. Pressman S. R, Maxim. B. (2019). Software Engineering: A Practitioner's Approach 9th Edition. McGraw Hill. ISBN 9780078022128.
17. Boehm, B.W. (2001). Software Engineering Economics. In: Broy, M., Denert, E. (eds) Pioneers and Their Contributions to Software Engineering. Springer, Berlin, Heidelberg. ISBN: 978-3-642-48354-7.
18. Pfleeger, S. L., & Atlee, J. M. (2010). Software Engineering: Theory and Practice. Pearson. ISBN-10: 0136061699
19. Boehm, B., & Turner, R. (2004). Balancing Agility and Discipline: A Guide for the Perplexed In: Ramamoorthy, C.V., Lee, R., Lee, K.W. (eds) Software Engineering Research and Applications. SERA 2003. Lecture Notes in Computer Science, vol 3026. Springer, Berlin, Heidelberg. ISBN: 978-3-540-21975-0
20. Humphrey, W. S. (1995). A Discipline for Software Engineering. Addison-Wesley

Professional. ISBN-10: 1572485965

21. Leffingwell, D., & Widrig, D. (2003). Managing Software Requirements: A Use Case Approach. Addison-Wesley Professional. ISBN-10: 032112247X

22. Thayer, R. H., & Dorfman, M. (2000). Software Requirements Engineering. Wiley-IEEE Computer Society Pr . ISBN-10: 0818677384

23.

24. Kenneth E. Kendall dan Julie E. Kendall (2014). Systems Analysis and Design. Pearson. ISBN-10: 013478555X. 2014

25. Roger S. Pressman S. R, Maxim. B. (2019). Software Engineering: A Practitioner's Approach 9th Edition. McGraw Hill. ISBN 9780078022128.

26. Klaus Pohl (2010). Requirements Engineering: Fundamentals, Principles, and Techniques. Springer. ISBN-10: 3642125778.

27. Suzanne Robertson & James Robertson (2013). Mastering the Requirements Process: Getting Requirements Right. Addison-Wesley Professional. ISSN-10: 0321815742

28. Steve McConnell. (1996). Rapid Development: Taming Wild Software Schedules. Microsoft Press. ISBN-10: 9781556159008

29. Daniel R. Windle. (2002). Software Requirements Using the Unified Process: A Practical Approach. Prentice Hall. ISBN-10: 0130969729.

30. Gunnar Overgaard & Karin Palmkvist. (2005). Applying Use Cases: A Practical Guide. Addison-Wesley Professional. ISBN-10: 0201708531.

31. Dean Leffingwell & Don Widrig. (2003). Managing Software Requirements: A Unified Approach. Addison-Wesley. ISBN-10: 0201615932

32. Benjamin M. Brothers (2011). The Art of Software Modeling. Auerbach Publications. ISBN-10: 1420044621

5. Daftar Bacaan

1. Sama seperti pada daftar referensi

6. JADWAL PERKULIAHAN DAN TOPIK BAHASAN

Pertemuan Ke-	TOPIK BAHASAN	BACAAN
1	a. Kontrak Perkuliahan, Perkenalan dan Penjelasan b. Pengenalan Rekayasa Perangkat Lunak	Kontrak Perkuliahan
2	a. Karakteristik perangkat lunak b. Komponen perangkat lunak	1-6

	c. Model perangkat lunak d. Fungsi dan peran dari software engineer	
3	a. Definisi SDLC b. Jenis-jenis SDLC	Idem
4	a. Observasi dan estimasi dalam perencanaan proyek b. Tujuan perencanaan proyek c. Manajemen proyek perangkat lunak yang efektif	Idem
5	a. Proses analisis kebutuhan b. Metode analisis kebutuhan c. Spesifikasi dan validasi kebutuhan	Idem
6	a. Perangkat bantu proses analisis kebutuhan b. Konsep dasar, Konteks, Proses, dan Prinsip Perancangan Perangkat Lunak; c. Isu mendasar dalam perancangan perangkat lunak	Idem
7	a. Alat bantu perancangan (DFD dan UML) b. Macam-macam diagram yang terdapat pada UML (Class Diagram, Use Case Diagram, Activity Diagram, Sequence Diagram)	Idem
8	**UTS**	
9	a. Konsep dalam User Interface b. Desain User Interface c. Prinsip Desain antarmuka (user experience, user guidance, user diversity)	Idem
10	a. Perencanaan dalam pengujian b. Proses testing: (black box testing, white box testing) c. Integration testing dan user testing d. Faults, Error dan Failures	Idem
11	Review Teknik Pengujian Perangkat Lunak dari proses testing	Idem
12	a. Pengujian unit b. Pengujian integrasi c. Pengujian sistem d. Debugging dan quality assurance	Idem
13	a. Quality assurance pada perangkat lunak b. Keamanan data akses	Idem
14	a. Definisi pemeliharaan perangkat lunak. b. Konsep Pemeliharaan Perangkat lunak	Idem
15	Teknik pemeliharaan perangkat lunak (Pemeliharaan	Idem

	korektif, pemeliharaan adaptif, pemeliharaan perfektif, pemeliharaan preventif)	
16	**UAS**	