Erdianti Wiga Putri Andini

13522053 – K1

TUGAS BACA

CHAPTER 3 HOLGER GAST

3.1. Inheritance

Inheritance dalam bahasa pemrograman berorientasi objek seperti Java sering dianggap sebagai alat yang powerful untuk penggunaan kembali kode dan fleksibilitas. Namun, penyalahgunaannya dapat menyebabkan masalah pemeliharaan, karena hirarki yang rumit dari kelas-kelas turunan dan kelas dasar dapat sulit untuk dinavigasi. Prinsip Substitusi Liskov (LSP) berfungsi sebagai pedoman penggunaan inheritance yang aman dan dapat diprediksi. Ini menekankan bahwa objek-objek dari kelas turunan harus dapat menggantikan (menjaga perilaku) objek-objek dari kelas dasar mereka.

Di Java, hubungan subtyping dideklarasikan menggunakan klausa "extends" dan "implements". Hubungan-hubungan ini memastikan bahwa kelas-kelas turunan dapat digunakan di mana pun kelas dasar mereka diharapkan. Sementara pendekatan Java terutama bergantung pada deklarasi eksplisit untuk subtyping, subtyping struktural dan subtyping perilaku menawarkan perspektif alternatif. Subtyping struktural berfokus pada struktur objek, sementara subtyping perilaku menekankan reaksi objek terhadap panggilan metode, melengkapi subtyping struktural dengan asersi tentang perilaku yang diharapkan.

LSP juga berlaku untuk mendefinisikan interface antara kelas dasar dan kelas turunannya. Dengan mempertimbangkan setiap kelas memiliki dua interface—satu untuk klien dan satu untuk kelas turunan—pengembang dapat merancang hirarki inheritance yang seimbang antara aksesibilitas dan enkapsulasi. Akses terproteksi memungkinkan kelas turunan mengakses fungsionalitas tertentu sambil mencegah akses tidak terbatas ke bagian dalam kelas dasar, mengurangi masalah kelas dasar yang rapuh.

Penggunaan yang efektif dari inheritance melibatkan pemisahan perilaku umum ke dalam kelas dasar. Dengan menangkap perilaku bersama, kelas dasar menyediakan infrastruktur yang dapat digunakan kembali untuk kelas-kelas turunan. Infrastruktur ini tidak hanya menawarkan fungsionalitas dasar, tetapi juga memperluas interface terproteksi untuk kelas-kelas turunan untuk memanipulasi internal dan mengimplementasikan mekanisme generik yang dapat disesuaikan oleh kelas-kelas turunan melalui pola Metode Template.

Inheritance juga berfungsi untuk mengabstraksi dan mengklasifikasikan entitas dalam sistem. Hiranarsi abstraksi menangkap perbedaan kasus hierarkis, di mana kelas-kelas dasar mewakili konsep yang lebih umum dan kelas-kelas turunan mewakili kasus-kasus khusus. Namun, penting untuk menghindari pengabstraksian yang berlebihan, karena setiap kelas menambah kompleksitas dan biaya pemeliharaan. Pengembang harus memprioritaskan memberikan abstraksi yang bermakna yang memberikan kontribusi signifikan pada fungsionalitas sistem sambil menjaga hirarki dangkal dan sempit.

Terakhir, inheritance dapat merepresentasikan perbedaan kasus sebagai objek. Pendekatan ini mengumpulkan kode yang terkait dengan setiap kasus, menjadikannya lebih mudah untuk dikelola dan diperluas. Namun, pertimbangan yang hati-hati diperlukan untuk menyeimbangkan manfaat dari kemampuan memperluas dengan overhead dalam

mengelola hierarki kelas. Pengembang harus bertujuan untuk abstraksi perilaku dan perbedaan kasus yang dapat diperluas untuk memastikan bahwa hirarki inheritance mereka tetap fleksibel dan dapat dipelihara.

Ketika pemrograman berorientasi objek pertama kali populer, salah satu daya tariknya adalah kemampuan untuk mengadaptasi fungsionalitas yang ada dengan mewarisi dan mengganti metode. Namun, pendekatan ini untuk pemrograman dengan perbedaan, meskipun pada awalnya menarik, sering menghasilkan kode yang rapuh dan rentan terhadap perubahan pada kelas dasar. Seiring waktu, para pengembang mengakui pentingnya antisipasi dan refaktorisasi dalam mencapai penggunaan kembali, menekankan perlunya identifikasi eksplisit dari perilaku umum.

Dalam kasus di mana kebutuhan untuk adaptasi minim, seperti menambah sedikit fungsionalitas tambahan ke dalam kelas-kelas internal, gagasan asli tentang pemrograman dengan perbedaan mungkin masih berguna. Namun, penting untuk menyadari bahwa penggunaan inheritance seperti ini mirip dengan inheritance implementasi, yang dapat meningkatkan secara signifikan kemungkinan kelas tersebut rusak dengan perubahan pada kelas dasar.

Preferensi untuk delegasi daripada inheritance secara luas diakui dalam pengembangan perangkat lunak. Delegasi menawarkan beberapa keuntungan, termasuk kontrol yang lebih baik atas fungsi mana yang diwariskan, menghindari keterikatan yang erat antara kelas-kelas, dan kemampuan untuk mengubah perilaku yang didelegasikan tanpa memengaruhi klien.

Inheritance memperkenalkan keterikatan yang erat antara kelas turunan dan kelas dasar, membuat penalaran tentang kode menjadi lebih kompleks dan menimbulkan tantangan untuk pemahaman dan pemeliharaan. Selain itu, penggunaan pengganti metode yang tidak terdisiplin dapat menghasilkan konsekuensi yang tidak diinginkan dan kode yang rapuh.

Untuk mengurangi masalah kelas dasar yang rapuh, pengembang harus menghindari akses langsung ke status kelas dasar, mendokumentasikan ketergantungan dan hubungan panggilan antara metode, menentukan metode yang dapat diganti dan mekanisme infrastruktur, dan menghindari asumsi tentang perilaku metode yang digantikan, terutama dalam kasus panggilan sendiri. Dengan mengikuti panduan ini, pengembang dapat membuat sistem perangkat lunak yang lebih kuat dan dapat dipelihara yang kurang mungkin rusak dengan perubahan pada kelas dasar.

3.2. Interfaces

Interface dalam Java menawarkan mekanisme fleksibel untuk memisahkan definisi metode dari implementasi konkretnya. Hal ini memungkinkan penggunaan ulang kode yang lebih baik, mempromosikan subtyping perilaku, dan meningkatkan pemeliharaan kode. Interface memungkinkan kelas untuk menjanjikan implementasi perilaku tertentu tanpa

membatasi hierarki pewarisan mereka, sehingga menawarkan pendekatan yang lebih fleksibel dibandingkan dengan pewarisan tunggal atau ganda.

Kekuatan interface juga dapat menimbulkan bahaya over-engineering. Desainer harus mempertimbangkan dengan cermat kebutuhan objek klien sebelum membuat interface baru. Interface yang tidak perlu dapat mengotori kode dan mempersulit pemahaman.

Interface sangat baik dalam menangkap perilaku umum yang ditunjukkan oleh objek yang berbeda di seluruh sistem. Dengan merancang interface dari sudut pandang konseptual daripada mengikatnya dengan implementasi tertentu, pengembang dapat memaksimalkan penggunaan kembali dan fleksibilitas kode.

Konsep interface ekstensi menangani tantangan evolusi API tanpa mengganggu kompatibilitas ke belakang. Interface ekstensi memungkinkan perluasan fungsionalitas secara bertahap sambil memastikan kompatibilitas dengan implementasi yang ada. Pendekatan ini menawarkan keseimbangan antara perluasan dan pemeliharaan, memungkinkan pengembang untuk beradaptasi dengan persyaratan yang berkembang tanpa mengorbankan stabilitas.

Interface memainkan peran penting dalam memisahkan subsistem dan mempromosikan fleksibilitas dalam desain perangkat lunak. Dengan menentukan interface yang mengemas perilaku atau pemberitahuan tertentu, pengembang dapat melindungi klien dari kompleksitas internal modul. Pemisahan ini memungkinkan perubahan implementasi modul tanpa memengaruhi kode klien.

Meskipun memiliki beberapa kelemahan, manfaat pemisahan melalui interface dapat melebihi biayanya, terutama dalam skenario di mana perubahan yang diharapkan terhadap sebuah modul sangat signifikan atau di mana modul dimaksudkan untuk diperluas oleh beberapa klien. Dalam kasus-kasus seperti itu, interface berfungsi sebagai cangkang pelindung di sekitar implementasi konkret, memastikan bahwa klien tetap tidak terpengaruh oleh modifikasi internal.

Interface penanda dan manajemen konstan menunjukkan penggunaan alternatif interface di luar abstraksi perilaku. Interface penanda, seperti Cloneable atau Serializable, bertindak sebagai flag yang melekat pada objek, sementara interface yang menentukan konstan mengumpulkan nilai-nilai terkait untuk kenyamanan dan konsistensi.

Ketika memutuskan antara inheritance dan interface, penting untuk mempertimbangkan kelebihan dan kekurangan masing-masing. Interface sangat baik dalam menangkap perilaku abstrak, memisahkannya dari detail implementasi, dan memungkinkan skema klasifikasi yang berbeda. Di sisi lain, inheritance lebih disukai untuk membuat infrastruktur umum dan menerapkan perbedaan kasus yang tetap atau terbatas.

Pada akhirnya, pilihan antara inheritance dan interface tergantung pada persyaratan dan kendala spesifik dari perangkat lunak yang dikembangkan. Pengembang harus menimbang kelebihan dan kekurangan dari setiap pendekatan dan menyesuaikan strategi mereka sesuai, bertujuan untuk keseimbangan yang mempromosikan fleksibilitas, pemeliharaan, dan kejelasan dalam kode sumber.