**Kapittel 10**

I dette kapittelet utforskes Design Patterns, som er gjenbrukbare løsninger på vanlige designproblemer i programvareutvikling. Her er en oppsummering av hovedpunktene i kapittelet:

1. Hva er Design Patterns?

* Design Patterns er dokumenterte løsninger på vanlige designproblemer som oppstår i programvareutvikling.
* De gir en felles terminologi og etablerte beste praksis for å løse spesifikke designutfordringer.

1. Typer av Design Patterns:

* Creational Patterns: Håndterer objektinstansiering og opprettelse av objekter på en fleksibel og generell måte.
* Structural Patterns: Handler om sammensetning av objekter og hvordan de organiseres for å danne større strukturer.
* Behavioral Patterns: Fokuserer på interaksjonen mellom objekter og definering av kommunikasjonsmønstre.

1. Eksempler på Design Patterns:

* Singleton Pattern: Sikrer at det bare finnes en enkelt instans av en klasse i systemet.
* Factory Pattern: Abstrakter objektinstansieringen ved å delegere opprettelsen til en egen fabrikkklasse.
* Observer Pattern: Gir en måte å oppdage og varsle om endringer i et objekt til andre avhengige objekter.
* Strategy Pattern: Definerer et sett med algoritmer og gjør det mulig å bytte mellom dem dynamisk.
* MVC (Model-View-Controller) Pattern: Deler applikasjonen inn i modell, visning og kontroller for å oppnå separat ansvar og bedre vedlikeholdbarhet.

1. Bruk av Design Patterns:

* Design Patterns er ikke en "one size fits all" -løsning, men bør brukes når de passer til det spesifikke designproblemet.
* De bidrar til å gjøre koden mer modulær, fleksibel og gjenbrukbar.
* Det er viktig å forstå Design Patterns grundig og bruke dem på en hensiktsmessig måte for å unngå overdesign eller unødvendig kompleksitet.

Boken utforsker flere Design Patterns og gir grundige eksempler, implementeringsdetaljer og diskusjoner om deres fordeler og ulemper. Forståelse av Design Patterns gir utviklere et verktøysett for å takle vanlige designproblemer og skape godt strukturert, fleksibel og vedlikeholdbar kode.

**Kapittel 11**

Kapittel 11 i boken "Agile Technical Practices Distilled: A Learning Journey in Technical Practices and Principles of Software Design" fokuserer på begrepene sammenheng (cohesion) og kobling (coupling) i programvaredesign. Her er en oppsummering av kapittelet:

Kapittelet starter med å definere kobling som graden av gjensidig avhengighet mellom programvarekomponenter. Høy kobling betyr at endringer i en komponent krever endringer i en annen komponent, mens lav kobling tillater endringer i en komponent uten å påvirke andre.

Flere "code smells" blir identifisert som indikatorer på mulig høy kobling, inkludert "Divergent Change", "Feature Envy", "Inappropriate Intimacy", "Message Chains", "Middleman" og "Shotgun Surgery".

Kapittelet diskuterer deretter ulike typer kobling:

1. Interaksjonskobling: Dette er den vanligste typen kobling og oppstår når en komponent kaller metoder fra en annen komponent. Forskjellige typer interaksjonskobling inkluderer "Message Coupling", "Data Coupling", "Stamp Coupling", "Control Coupling", "External Coupling", "Common Coupling" og "Content Coupling".
2. Arvekobling: Denne typen kobling oppstår når en klasse arver fra en annen klasse. Endringer i superklassen kan påvirke alle underklasser, noe som gjør arvekobling sterk.

Demeters lov introduseres som et prinsipp for å unngå kobling mellom komponenter. Den sier at et objekt bare skal kalle metoder til seg selv, dens attributter, parameterobjekter, objekter som er opprettet av metoden, eller objekter som er lagret i globale variabler.

Deretter utforsker kapittelet begrepet sammenheng, som måler hvor sterkt relaterte og fokuserte ansvarsområdene til en programvaremodul er. Høy sammenheng indikerer at en komponent implementerer en enkelt atferd eller ansvar effektivt, mens lav sammenheng resulterer i spredt atferd over flere komponenter.

Indikatorer på mulig lav sammenheng blir nevnt, inkludert "Data class", "Lazy class", "Middleman", "Primitive obsession" og "Shotgun Surgery". På den annen side er indikatorer på for mye sammenheng "Data clumps", "Divergent change", "Duplication", "Large class", "Long method" og "Long parameter list".

To hovedkategorier av sammenheng blir diskutert:

1. Klasse-sammenheng: Dette refererer til sammenhengen på klassenivå og kan være ideell, blandet-rolle, blandet-domene eller blandet-instans.
2. Metode-sammenheng: Dette fokuserer på sammenhengen til individuelle metoder og inkluderer funksjonell sammenheng, sekvensiell sammenheng, kommunikativ/informasjonsmessig sammenheng, prosedyremessig sammenheng, tidsmessig sammenheng, logisk sammenheng og tilfeldig sammenheng.

Kapittelet kobler også "code smells" med sammenheng og kobling, og gir en tabell som grupperer "code smells" basert på deres forhold til overskudd av kobling eller sammenheng.

Object Calisthenics og andre heuristikker nevnes som verktøy for å unngå "code smells" og opprettholde god sammenheng og kobling i koden.

Kapittelet avsluttes med å understreke viktigheten av refaktorering i situasjoner der koden viser lav sammenheng eller høy kobling, sammen med andre "code smells" og brudd på object calisthenics-regler.

Vennligst merk at denne oppsummeringen er basert på informasjonen du ga, og den dekker kanskje ikke alle detaljer og nyanser som diskuteres i det faktiske kapittelet i boken. For en full forståelse anbefales det å henvise til den opprinnelige kilden.

**Kap.12**

I dette kapittelet introduseres "Solid Principles ++", som er en utvidet versjon av de klassiske SOLID-prinsippene i objektorientert design. Her er en oppsummering av hovedpunktene i kapittelet:

1. Single Responsibility Principle (SRP):

* En klasse bør ha bare én grunn til å endre seg.
* Bruk "High Cohesion, Low Coupling" for å oppnå SRP.

1. Open-Closed Principle (OCP):

* Softwareentiteter skal være åpne for utvidelse, men lukket for endring.
* Bruk abstraksjoner og polymorfisme for å oppnå OCP.

1. Liskov Substitution Principle (LSP):

* Objekter av en subtype skal kunne erstatte objekter av sin basistype uten å påvirke programmet korrekthet.
* Unngå designbrudd ved å forstå og overholde LSP.

1. Interface Segregation Principle (ISP):

* Klienter skal ikke tvinges til å avhenge av grensesnitt de ikke bruker.
* Del store og generaliserte grensesnitt opp i mindre og mer spesifikke grensesnitt.

1. Dependency Inversion Principle (DIP):

* Høy-nivåmoduler skal ikke avhenge av lav-nivåmoduler. Begge skal avhenge av abstraksjoner.
* Benytt deg av avhengighetsinjeksjon og inversjon av kontroll for å oppnå DIP.

I tillegg til SOLID-prinsippene blir følgende konsepter introdusert som en del av "Solid Principles ++":

1. Composition Over Inheritance:

* Foretrekk sammensetning (composition) fremfor arv (inheritance) når det er mulig.
* Sammensetning gir større fleksibilitet og løs kobling.

1. Don't Repeat Yourself (DRY):

* Unngå duplisering av kode og logikk.
* Gjenbruk eksisterende kode og abstraksjoner for å eliminere duplisering.

1. Law of Demeter (LoD):

* En klasse skal bare ha kjennskap til sine umiddelbare naboer, ikke til hele systemet.
* Reduser koblingen mellom klasser for å forbedre vedlikeholdbarhet og fleksibilitet.

1. Single Level of Abstraction (SLA):

* Metoder og klasser skal ha en enkelt abstraksjonsnivå.
* Hold abstraksjonsnivået konsistent for bedre lesbarhet og forståelse.

I boken utforskes disse prinsippene grundig med forklaringer, eksempler og diskusjoner om deres praktiske anvendelse. Gjennom å bruke disse prinsippene kan utviklere oppnå bedre design og mer vedlikeholdbar kode.

**Kapittel 13**

Kapittel 13 av boken "Design VII: The Agile Technical Practices Distilled" tar for seg viktigheten av tekniske praksiser i den smidige utviklingsprosessen. Her er en oppsummering av kapittelet:

1. Kontinuerlig integrasjon (Continuous Integration): Kapittelet starter med å introdusere kontinuerlig integrasjon, som er en praksis der utviklere integrerer og tester kodeendringer regelmessig. Dette bidrar til å oppdage og fikse problemer tidlig i utviklingsprosessen, og sikrer at koden alltid er i en fungerende tilstand.
2. Testdrevet utvikling (Test-Driven Development - TDD): TDD er en praksis der utviklere skriver tester før de implementerer selve koden. Dette hjelper til med å sikre at koden oppfyller de ønskede kravene og fungerer som forventet. TDD bidrar til å forbedre kvaliteten på koden og gir økt tillit til endringene som gjøres.
3. Refaktorering (Refactoring): Refaktorering innebærer å restrukturere koden uten å endre dens observerbare atferd. Denne praksisen bidrar til å forbedre kodekvaliteten, gjøre koden mer lesbar og lettere å vedlikeholde. Ved kontinuerlig refaktorering opprettholdes kodebasen i en sunn tilstand og reduserer teknisk gjeld.
4. Parprogrammering (Pair Programming): Parprogrammering involverer to utviklere som samarbeider om å skrive kode sammen. Dette gir fordeler som kunnskapsdeling, raskere feilretting, bedre design og økt kvalitet. Parprogrammering kan også bidra til å forbedre samarbeidet og kommunikasjonen i utviklingsteamet.
5. Enhetstesting (Unit Testing): Enhetstesting er en praksis der utviklere skriver automatiserte tester for individuelle komponenter av koden, kalt enheter. Dette bidrar til å sikre at hver enhet fungerer som forventet og oppfyller kravene. Enhetstester gjør det enklere å identifisere og fikse feil tidlig i utviklingsprosessen.
6. Smidig arkitektur (Agile Architecture): Kapittelet fremhever betydningen av å ha en smidig tilnærming til arkitektur. Det innebærer å fokusere på evolusjonær design og kontinuerlig forbedring av arkitekturen basert på endringer i krav og forretningsbehov. Smidig arkitektur bidrar til å sikre at systemet er fleksibelt, skalerbart og enkelt å vedlikeholde.

Oppsummert fokuserer kapittel 13 på betydningen av tekniske praksiser i den smidige utviklingsprosessen. Kontinuerlig integrasjon, testdrevet utvikling, refaktorering, parprogrammering, enhetstesting og smidig arkitektur er alle viktige praksiser som bidrar til å forbedre kvaliteten på koden, øke produktiviteten og sikre suksess i smidige prosjekter.