Utmaningarna som arkitekturarbetet tar sig an

- **Komplexitet**: Hantering av teknisk och organisatorisk komplexitet i mjukvarusystem.
- **Krav**: Balans mellan olika krav, som funktionella och icke-funktionella krav.
- **Ändringar**: Anpassning till förändrade krav och teknologiska trender under utvecklingscykeln.
- **Integrering**: Samordning av olika komponenter och system för att säkerställa att de fungerar tillsammans.

Varför är arkitekturdesign viktigt?

- **Struktur**: Ger en övergripande struktur och vägledning för systemets utveckling.
- **Kvalitet**: Påverkar systemets kvalitet, prestanda och underhållbarhet.
- **Riskhantering**: Identifierar och hanterar risker tidigt i utvecklingsprocessen.
- **Kommunikation**: Underlättar kommunikation mellan intressenter och utvecklingsteam.

Skillnader mellan design på en hög och en låg nivå

- **Hög nivå**: Fokuserar på systemets övergripande arkitektur, komponenter och deras relationer. Inkluderar designbeslut som påverkar hela systemet.
- **Låg nivå**: Fokuserar på specifika implementationer och detaljer inom enskilda komponenter. Inkluderar programmering och detaljerad design.

Exempel på faktorer som bidrar till komplexitet

- **Integration av olika teknologier**: Att kombinera olika ramverk och plattformar kan leda till komplexitet.
- **Många användarkrav**: Många intressenter med olika krav kan göra systemet mer komplext.
- **Skalbarhet**: Att designa system som ska kunna hantera ökad användning eller data kan öka komplexiteten.
- **Tekniska begränsningar**: Prestanda, säkerhet och tillförlitlighet kan lägga till komplexitet.

Hantering av komplexitet

- **Abstraktion**: Användning av abstrakta modeller för att förenkla komplexa system.
- **Modularisering**: Dela upp systemet i mindre, hanterbara moduler som kan utvecklas och testas oberoende.
- **Standardisering**: Användning av standarder och ramverk för att minska variation och osäkerhet.
- **Iterativ utveckling**: Använda iterativa metoder för att gradvis förfina och förbättra systemet.

Vad är en "architecture runway"?

En "architecture runway" är en struktur som tillhandahåller de tekniska komponenter och infrastrukturer som krävs för att stödja utvecklingen av funktionalitet i ett mjukvarusystem. Den syftar till att hantera problem relaterade till teknisk skuld, bristande infrastruktur och planering av framtida funktioner. Den ska användas kontinuerligt under utvecklingsprocessen för att säkerställa att systemet kan utvecklas smidigt utan att begränsas av arkitektoniska hinder.

Abstraktion, hierarki, modularitet och inkapsling

Dessa begrepp är centrala för att hantera komplexitet i systematisk nedbrytning:

- **Abstraktion**: Gör det möjligt att dölja detaljer och fokusera på det väsentliga.
- **Hierarki**: Organiserar systemet i nivåer för att hantera komplexitet.
- **Modularitet**: Delar upp systemet i moduler för att förbättra hanterbarheten och återanvändningen.
- **Inkapsling**: Döljer interna detaljer och begränsar åtkomst till komponenters data, vilket minskar beroenden.

Exempel: En programvara för e-handel kan ha en modul för användarhantering, en för produktkatalog och en för betalningshantering. Varje modul kan utvecklas och testas separat, vilket förenklar utvecklingsprocessen.

Elements, elements interaction design och elements internal design

- **Elements**: De grundläggande byggstenarna i systemet, som klasser och komponenter.
- **Elements interaction design**: Hur dessa element interagerar med varandra och med användarna.
- **Elements internal design**: Detaljerad design och implementation av varje element.

Likheter: Alla tre aspekter bidrar till att definiera systemets struktur och beteende. Skillnader: Fokus på interaktioner (design) kontra interna implementationer (internal design).

Exempel på arkitekturdesignsteg (architectural drivers)

- **Kravspecifikationer**: Funktionella och icke-funktionella krav som påverkar designen.
- **Prestanda**: Krav på hastighet och effektivitet kan driva val av teknologier och arkitektur.
- **Säkerhet**: Krav på datasäkerhet kan påverka systemets design och implementering.
- **Underhållbarhet**: Behov av lätt underhåll och uppdateringar kan styra arkitekturen.

UML-diagram för att fånga arkitektur

1. **Klassdiagram**: Visar systemets klasser och deras relationer.
2. **Sekvensdiagram**: Illustrerar interaktioner mellan objekt över tid.
3. **Användningsfallsdiagram**: Definierar användarinteraktioner och systemets funktionalitet.
4. **Komponentdiagram**: Visar systemets komponenter och deras beroenden.
5. **Aktivitetsdiagram**: Visar flödet av aktiviteter och processer i systemet.
6. **Tillståndsdiagram**: Representerar tillstånd och övergångar mellan dem i systemet.

Dessa diagram hjälper till att fånga olika aspekter av systemets arkitektur och dokumentera viktiga arkitekturbeslut.

Steg i Attribute Driven Design (ADD)

1. **Planering**: Identifiera arkitekturens attribut och definiera krav och begränsningar.
2. **Attributdefinition**: Specificera och prioritera de viktiga attributen för systemet (t.ex. prestanda, säkerhet).
3. **Designstrategier**: Välj designstrategier och mönster som stödjer de definierade attributen.
4. **Arkitekturella beslut**: Ta beslut om arkitekturens struktur och komponenter baserat på de valda strategierna.
5. **Verifikation**: Granska och verifiera att designen uppfyller de definierade attributen och kraven.
6. **Dokumentation**: Dokumentera designbeslut och deras motiveringar.

Inkrementell och iterativ process

ADD är inkrementell eftersom den tillåter stegvis utveckling av arkitekturen, vilket innebär att designen kan utvidgas med nya funktioner eller attribut vid behov. Den är iterativ eftersom den kräver flera cykler av utvärdering och förbättring av designen, vilket gör det möjligt att justera och optimera designbeslut över tid baserat på feedback och nya insikter.

Förutsättningar i ADD

1. **Affärsmål**: Förstå organisationens mål och syften.
2. **Intressenters krav**: Identifiera krav från alla relevanta intressenter.
3. **Tekniska begränsningar**: Känna till tekniska begränsningar och möjligheter.
4. **Systemkontext**: Definiera systemets kontext och gränssnitt.
5. **Existerande system**: Beakta eventuella befintliga system och deras egenskaper.
6. **Prestanda och skalbarhet**: Ta hänsyn till prestanda- och skalbarhetskrav.
7. **Säkerhet**: Definiera säkerhetskrav och -risker.
8. **Underhållbarhet**: Beakta krav på systemets underhåll och vidareutveckling.

Skillnad mellan constraint (villkor/avgränsning) och constraint (angelägenhet)

- **Constraint (villkor/avgränsning)**: En begränsning som definierar vad som är möjligt eller omöjligt i designprocessen. Till exempel kan det vara tekniska eller resursmässiga begränsningar.
- **Constraint (angelägenhet)**: En prioritering som beskriver hur viktigt ett visst krav eller attribut är för projektets framgång. Till exempel kan säkerhet vara en högre angelägenhet än prestanda i vissa sammanhang.

Design purpose (syfte)

Design purpose beskriver det övergripande målet eller syftet med designprocessen. Det anger vilket problem designen ska lösa och vilket värde den ska skapa. Tre exempel på design purpose som förändrar fokus för designprocessen är:

1. **Användarcentrerad design**: Fokuserar på användarupplevelsen och behov.
2. **Prestandafokus**: Prioriterar systemets hastighet och effektivitet.
3. **Säkerhetsfokus**: Lägger stor vikt vid att skydda data och användarinformation.

Säkerställande av iterationens syfte

Iterationens syfte säkerställs genom att definiera specifika mål för varje iteration och genomföra utvärderingar för att se om dessa mål uppnåtts. Om syftet inte uppnåtts kan det leda till:

- Omvärdering av designbeslut.
- Justering av krav och prioriteringar.
- Behov av ytterligare iterationer för att nå målen.

Specifika designaktiviteter i ADD

1. **Attributanalys**: Identifiera och prioritera systemattribut.
2. **Designval**: Välja lämpliga designstrategier och mönster.
3. **Utvärdering**: Granska designbesluten mot attributen och kraven.

4. **Feedback**: Samla feedback från intressenter och justera designen.

Dessa aktiviteter förhåller sig till varandra genom att de bygger på tidigare steg och iterativt förbättrar designen. De leder fram till en välgrundad och dokumenterad arkitektur som uppfyller systemkraven och attributen.

Principer för Beslutsfattande

För att minimera risken för att individuella uppfattningar påverkar beslut kan du använda följande principer i praktiken:

1. **Involvera flera perspektiv**: Skapa ett mångsidigt team där olika erfarenheter och kunskaper representeras.
2. **Datadrivet beslutsfattande**: Använd objektiva data och analyser för att informera beslut snarare än subjektiva känslor.
3. **Strukturerad process**: Använd en tydlig beslutsfattande process med definierade steg och kriterier för att utvärdera alternativ.
4. **Öppen kommunikation**: Främja en kultur där alla kan uttrycka sina åsikter och ifrågasätta beslut, vilket minskar risken för grupptänkande.
5. **Feedbackloopar**: Implementera mekanismer för att utvärdera resultaten av beslut och lära av dem för framtida beslut.

Taktik och Mönster

Taktik och mönster hänger ihop genom att taktik är specifika tekniker eller metoder som används för att uppnå vissa mål inom ramen för ett designmönster. Mönster erbjuder övergripande lösningar på återkommande problem, medan taktik beskriver hur dessa mönster kan implementeras effektivt.

Rationellt vs. Begränsat Rationellt Beslutsfattande

- **Rationellt beslutsfattande**: Antar att beslutsfattare har all information, kan analysera alla alternativ och alltid väljer det bästa alternativet för att maximera nyttan.
- **Begränsat rationellt beslutsfattande**: Erkänner att beslutsfattare har begränsad information, begränsad förmåga att bearbeta information och ofta nöjer sig med ett tillfredsställande snarare än det optimala beslutet. Det innebär att beslut fattas inom rimliga tidsramar och resurser.

Designmönster vs. Arkitekturmönster

- **Designmönster**: Fokuserar på specifika lösningar på problem i kod eller i programvarudesign. De används ofta på en mer detaljerad nivå i programvarans konstruktion.
- **Arkitekturmönster**: Fokuserar på högre nivåer av systemdesign och beskriver övergripande strukturer och relationer mellan systemets komponenter. De adresserar hur systemets olika delar interagerar och organiseras.

Mönster för Client-Server och Peer-to-Peer

- **Client-Server**: I detta mönster kommunicerar klienter (användare) med en central server som tillhandahåller resurser och tjänster. Servern hanterar datalagring, autentisering och åtkomstkontroll, vilket centraliserar systemets resurser.

- **Peer-to-Peer**: I detta mönster fungerar varje enhet både som klient och server. Användare kan dela resurser direkt med varandra utan behov av en central server. Detta skapar decentraliserade nätverk där alla enheter är lika viktiga.

Publish-Subscribe Mönster

Publish-Subscribe mönstret karakteriseras av att producenter (publishers) publicerar meddelanden eller händelser utan att veta vilka konsumenter (subscribers) som kommer att ta emot dem. Konsumenter prenumererar på specifika ämnen eller typer av meddelanden och får endast meddelanden som de är intresserade av. Detta möjliggör lös koppling mellan producenter och konsumenter, vilket ökar flexibiliteten i systemet.

Arbeta med Kvalitetskrav utan Färdigt Träd

Om du arbetar med ett kvalitetskrav eller en angelägenhet utan ett färdigt träd med taktiker och mönster, kan du:

1. **Identifiera relevanta attribut**: Definiera vad som är viktigt för kvalitetskravet (t.ex. prestanda, säkerhet).
2. **Forska och samla in mönster**: Studera befintliga designmönster och taktiker som kan adressera de identifierade attributen.
3. **Engagera teamet**: Diskutera och brainstorma med teamet för att hitta kreativa lösningar och strategier.
4. **Prototypa**: Utveckla prototyper för att testa olika taktiker och mönster för att se vilka som fungerar bäst.

Jämförelse mellan Mönster för Fysisk Arkitektur och Allokeringsstruktur

- **Fysisk arkitektur**: Handlar om hur hårdvarukomponenter (servrar, nätverk, datacenter) är organiserade och hur de interagerar för att stödja systemet. Mönster här inkluderar centraliserad, distribuerad, och redundanta system.

- **Allokeringsstruktur**: Fokuserar på hur källkod och resurser är organiserade inom programmet. Mönster här inkluderar modulära strukturer, paketorganisering och bibliotekshantering.

Skillnaden ligger i att fysisk arkitektur handlar om den faktiska implementeringen av hårdvara och nätverk, medan allokeringsstruktur handlar om hur koden och dess komponenter struktureras för att optimera prestanda och underhåll.

Referensarkitektur

Vad är en referensarkitektur? En referensarkitektur är en standardiserad, generaliserad beskrivning av en typisk systemarkitektur som används som en vägledning för design och implementering av specifika system. Den fungerar som en modell som definierar best practices, standarder och teknologier för att lösa specifika problem i en viss domän.

Roll i arkitekturdesignprocessen Referensarkitekturer fyller flera roller i arkitekturdesignprocessen:

1. **Vägledning**: Ger en utgångspunkt för designbeslut och hjälper till att säkerställa att alla komponenter fungerar tillsammans.

2. **Standardisering**: Främjar användning av gemensamma standarder och teknologier vilket underlättar integration och interoperabilitet.
3. **Effektivisering**: Genom att återanvända beprövade mönster kan designprocessen bli mer effektiv och snabbare.

Delar av en Referensarkitektur

En referensarkitektur brukar bestå av:

1. **Komponenter**: Identifierade och definierade delar som ingår i arkitekturen (t.ex. servrar, databaser, nätverkskomponenter).
2. **Relationer**: Beskrivning av hur komponenterna interagerar med varandra.
3. **Tjänster och gränssnitt**: Specification av de tjänster som erbjuds av komponenterna och hur de kommunicerar.
4. **Mönster och taktiker**: Definierade mönster och strategier som kan tillämpas i designen.
5. **Krav och riktlinjer**: Dokumentation av funktionella och icke-funktionella krav som arkitekturen ska uppfylla.

Taktiker och Mönster för Referensarkitekturer

Taktiker och mönster spelar en avgörande roll i referensarkitekturer genom att:

1. **Erbjuda lösningar**: De tillhandahåller beprövade lösningar för vanliga problem och utmaningar i systemdesign.
2. **Främja återanvändning**: Mönster och taktiker kan användas i flera olika projekt, vilket sparar tid och resurser.

Arkitektur och Återanvändning

Arkitektur spelar en viktig roll för design med och för återanvändning genom att:

1. **Skapa en struktur**: En väldefinierad arkitektur gör det lättare att återanvända komponenter och mönster i olika projekt.
2. **Möjliggöra anpassning**: Genom att definiera variabilitet och kommonalitet kan arkitekturen stödja både specifika och generella krav.

Variabilitet och Kommonalitet

Variabilitet (variability) refererar till möjligheten att anpassa eller variera en produktlinje för att möta specifika behov eller krav.

Kommonalitet (commonality) refererar till de gemensamma egenskaperna och funktionerna i en produktlinje som kan återanvändas mellan olika produkter.

Roll i Design för Återanvändning

Variabilitet och kommonalitet är avgörande för design för återanvändning eftersom:

1. **Flexibilitet**: Variabilitet tillåter anpassning till specifika behov medan kommonalitet möjliggör återanvändning av gemensamma komponenter.

2. **Effektivitet**: Genom att identifiera och separera variabla och gemensamma delar kan utvecklingsprocessen bli mer effektiv.

Mjukvaruproduktlinje och Arkitektur

En **mjukvaruproduktlinje** är en samling relaterade mjukvaruprodukter som delar en gemensam kärnarkitektur och uppsättning komponenter. Arkitekturen spelar en central roll i att definiera:

1. **Gemensamma komponenter**: Identifiera vilka delar som kan återanvändas mellan produkterna.
2. **Variabilitetsstrategier**: Definiera hur produkter kan anpassas och varieras baserat på gemensamma element.

Samspel mellan Taktiker och Mönster

Taktiker och mönster samspelar genom att taktiker implementerar de lösningar som definieras av mönster. Mönster ger en övergripande struktur, medan taktiker detaljerar specifika metoder för att uppnå de mål som anges i mönstren.

Designkoncept i Arkitekturdesignprocessen (ADD)

Designkoncept används i arkitekturdesignprocessen för att:

1. **Styra beslutsfattande**: Ge vägledning om hur specifika designbeslut ska fattas baserat på konceptuella modeller.
2. **Förenkla komplexitet**: Erbjuda abstraherade perspektiv för att hantera komplexiteten i design och implementering.

Taktikträd

Ett **taktikträd** är en hierarkisk representation av de taktiker som kan användas för att uppnå specifika mål i designen. Det är uppbyggt av:

1. **Rotnod**: Representerar det övergripande målet.
2. **Underordnade noder**: Varje nod representerar en specifik taktik eller metod för att uppnå målet.
3. **Anslutningar**: Visar relationer mellan taktiker och hur de kan kombineras för att uppnå större mål.

Taktikträdet används för att organisera och visualisera beslut kring taktiker, vilket hjälper till att identifiera de mest lämpliga metoderna för ett givet designproblem.

ATAM (Architecture Tradeoff Analysis Method)

Vad är ATAM? ATAM är en metod för att utvärdera och analysera mjukvaruarkitektur med fokus på att identifiera och hantera arkitektoniska trade-offs. Den syftar till att förbättra kvaliteten på arkitekturen och minska riskerna i utvecklingsprocessen.

Syften med ATAM

1. **Identifiera risker**: Hitta potentiella risker och problem tidigt i designprocessen.
2. **Prioritera krav**: Hjälpa till att prioritera funktionella och icke-funktionella krav.
3. **Utforska trade-offs**: Analysera kompromisser mellan olika arkitektoniska beslut.

4. **Dokumentera beslut**: Skapa en tydlig dokumentation av arkitekturella beslut och dess underliggande skäl.

Tillvägagångssätt i en ATAM-utvärdering

Faser i ATAM

1. **Förberedelsefas**
    - **Syfte**: Förstå projektets kontext, mål och krav.
    - **Gruppsammansättning**: Inkluderar intressenter som arkitekter, utvecklare, projektledare och användare.
    - **Fokus**: Identifiera och förstå arkitekturens syften och målsättningar.
    - **Arbetssätt**: Genomföra en workshop där målen och krav diskuteras samt en preliminär analys av systemets arkitektur görs.
2. **Utvärderingsfas**
    - **Syfte**: Genomföra en detaljerad analys av arkitekturen för att identifiera och utvärdera trade-offs.
    - **Gruppsammansättning**: Samma som i förberedelsefasen, med fokus på att ha en bred representation av intressenter.
    - **Fokus**: Utvärdera specifika arkitekturella beslut och deras inverkan på systemets kvalitet.
    - **Arbetssätt**: Använda checklistor och scenarier för att identifiera och diskutera risker och trade-offs, samt skapa en rapport som sammanfattar resultaten.

Utvärdering av Arkitekturdesign med Checklistor

Checklistor kan användas för att systematiskt utvärdera arkitekturdesign genom att:

1. **Standardisera utvärderingen**: Säkerställa att alla viktiga aspekter av designen beaktas.
2. **Identifiera brister**: Hitta potentiella problem eller förbättringsområden i arkitekturen.
3. **Föreslå åtgärder**: Ge rekommendationer baserat på identifierade brister och risker.

Förenklad ATAM

Vad är en "förenklad" ATAM? En förenklad ATAM är en version av metoden som kan genomföras på en dag och fokuserar på de mest kritiska delarna av utvärderingen.

- Den innebär snabbare identifikation av intressenter, krav och scenarier, och betonar mer direkt diskussion och prioritering av risker och trade-offs.

ATAM och ADD

ATAM relaterar till ADD (Attribute Driven Design) genom att båda metoderna fokuserar på att förbättra arkitekturens kvalitet. ATAM används för att analysera och utvärdera arkitekturen, medan ADD ger en struktur för att designa arkitekturen baserat på specifika attribut och krav.

Analys vs Utvärdering

Skillnaden mellan analys och utvärdering

- **Analys**: En mer teknisk och detaljerad granskning av arkitekturen som syftar till att identifiera potentiella problem och risker.
- **Utvärdering**: En mer övergripande bedömning av arkitekturens kvalitet och dess förmåga att uppfylla krav, vilket ofta involverar flera intressenter.

Alternativa Tillvägagångssätt för Bedömning och Utvärdering

1. **Kvalitativa metoder**: Intervjuer och workshops för att få insikter från intressenter.
2. **Kvantitativa metoder**: Mätningar och analyser av prestanda och andra mätbara kvaliteter.
3. **Formella metoder**: Användning av matematiska modeller och formella specifikationer för att verifiera designbeslut.

Steg av ATAM för Analys och Utvärdering

- **Analys**: Görs under förberedelsefasen där krav och mål diskuteras samt i utvärderingsfasen där scenarier och risker analyseras.
- **Utvärdering**: Sker under utvärderingsfasen där beslut tas baserat på analyser och insikter från diskussioner med intressenter.

Teknisk Skuld

Vad är teknisk skuld? Teknisk skuld avser de kompromisser och avvikelser från bästa praxis som görs under utvecklingsprocessen för att uppnå kortsiktiga mål. Detta kan leda till ökad komplexitet och kostnader på lång sikt.

Hur uppkommer teknisk skuld? Teknisk skuld uppstår ofta när team gör snabba lösningar eller genvägar för att möta tidsfrister, budgetbegränsningar eller krav från intressenter, vilket kan leda till en försämrad kodkvalitet eller arkitektur.

Vanliga orsaker till teknisk skuld

1. **Tidsbrist**: Utvecklare kanske prioriterar snabba lösningar framför mer hållbara lösningar.
2. **Otydliga krav**: Oklarhet kring projektkrav kan leda till felaktiga implementeringar.
3. **Brister i testning**: Otillräcklig testning kan leda till fel som senare måste åtgärdas.

Arkitektur Skuld

Vad är arkitektur skuld? Arkitektur skuld är en specifik typ av teknisk skuld som uppstår när arkitekturen för ett system inte längre stödjer de nuvarande eller framtida behoven på ett effektivt sätt.

Exempel på arkitektur skuld:

1. **Monolitisk arkitektur**:
    - **Orsak**: Ursprungligen utvecklad för ett litet system men växte utan refaktorering.
    - **Framtida problem**: Svårigheter med skalning och underhåll.
2. **Otydliga gränssnitt**:
    - **Orsak**: Snabb utveckling utan att definiera tydliga API:er.
    - **Framtida problem**: Komplikationer med integration av nya funktioner och tjänster.

3. **Bristande säkerhet**:
    - **Orsak**: Förbisedd säkerhet i designfasen för att spara tid.
    - **Framtida problem**: Ökad risk för säkerhetsintrång och dataläckor.

Arkitekturerosion vs Arkitekturavdrift

Skillnad mellan arkitekturerosion och arkitekturavdrift:

- **Arkitekturerosion**: Innebär en gradvis försämring av arkitekturens kvalitet över tid, ofta på grund av bristande underhåll eller teknisk skuld.
- **Arkitekturavdrift**: Refererar till en avvikelse från den ursprungliga arkitekturen på grund av nya krav eller förändringar, vilket kan leda till en inkonsekvent och svårhanterad arkitektur.

Förhållande till arkitekturskuld: Både erosion och avdrift kan bidra till ökad arkitekturskuld, vilket gör det svårare att upprätthålla en hållbar och effektiv arkitektur.

Beslutsfattande om Arkitekturskuld

Hur avgör man om ett beslut genererar arkitekturskuld? För att avgöra om ett beslut skapar arkitekturskuld behöver man:

1. **Förstå projektets långsiktiga mål**.
2. **Känna till arkitekturens nuvarande tillstånd och krav**.
3. **Bedöma konsekvenserna av beslutet på systemets framtida underhåll och skalbarhet**.

Refaktorisering

Vad innebär refaktorisering på arkitekturnivå? Refaktorisering innebär att förbättra strukturen och designen av befintlig arkitektur utan att förändra dess funktionalitet. Detta kan inkludera att omorganisera komponenter, förbättra gränssnitt eller uppgradera teknologier för att stödja nya krav.

Hantering av Arkitekturskuld

När och hur ofta bör man analysera och hantera arkitekturskuld? Arkitekturskuld bör analyseras och hanteras kontinuerligt under hela utvecklingsprocessen, speciellt i samband med större uppdateringar, iterationer och efter avslutade sprintar.

Acceptabla och Oacceptabla Situationer för Arkitekturskuld

Förståeliga situationer:

- När tidspress kräver snabba beslut för att möta kritiska deadlines.

Oacceptabla situationer:

- När grundläggande säkerhets- eller prestandakrav ignoreras för kortsiktiga vinster.

Svårighetsgrad av Skuld

Vilken typ av skuld är svårast att hantera? Arkitekturerosion är ofta svårare att hantera eftersom det är en gradvis process som kan leda till allvarliga konsekvenser över tid och därmed göra det svårare att

identifiera och åtgärda problem i tid.

ISO 42010

Vilken roll fyller ISO 42010? ISO 42010 är en standard för att definiera och dokumentera arkitektur i mjukvarusystem. Den hjälper till att skapa en gemensam förståelse för arkitekturen och dess komponenter samt att tydliggöra krav och intressenters behov.

Dokumentationsteknik för Arkitekturbeslut

Architecture Decision Records (ADR):

- **Översikt**: ADR är en teknik för att dokumentera viktiga arkitekturbeslut och deras motiveringar.
- **Innehåll**:
  - **Beslut**: Vad beslutet handlar om.
  - **Kontext**: Vilka omständigheter som ledde till beslutet.
  - **Konsekvenser**: Vilka effekter beslutet kan ha på systemet och framtida utveckling.

Dokumentera Kvalitetsattribut

Strategidokumentation för ett kvalitetsattribut:

- **Delavsnitt**:
  1. **Syfte och mål**: Vad man vill uppnå med kvalitetsattributet.
  2. **Beskrivning av attributet**: Vad attributet innebär och dess betydelse för systemet.
  3. **Mått och standarder**: Hur attributet kommer att mätas och utvärderas.
  4. **Risker och konsekvenser**: Vad kan gå fel och vilka risker finns det med att implementera eller inte implementera detta attribut.

Modeller för Arkitekturdokumentation

Typer av modeller:

1. **Strukturella modeller**: Beskriver systemets komponenter och deras relationer.
2. **Beteendemodeller**: Visar hur systemet reagerar på olika stimuli och scenarier.
3. **Interaktionsmodeller**: Fokuserar på hur användare och systemet interagerar.

Användning av Analytiska Modeller

Analytiska modeller kan användas för:

- Att utvärdera prestanda, skalbarhet och andra kvantitativa aspekter av arkitekturen.
- Att förutsäga systembeteende under olika belastningar.

Argumentationsmodell

Vad är en argumentationsmodell? En argumentationsmodell är en strukturerad representation av argument som stöder eller motsätter sig ett beslut. Den används för att:

- Klargöra rationale bakom arkitekturbeslut.
- Utvärdera för- och nackdelar med olika alternativ.

- Underlätta diskussioner och beslut bland intressenter.

API

Vad är ett API? Ett API (Application Programming Interface) är en uppsättning regler och protokoll som gör det möjligt för olika programvaror att kommunicera med varandra. Det definierar hur olika programkomponenter ska interagera.

Drivkrafter för Ökat Beroende av API:er

Vilka drivkrafter leder till ökat beroende av API:er i utvecklingsprojekt?

1. **Integration**: Behov av att koppla samman olika system och tjänster.
2. **Modularitet**: Främjar en modulär utveckling där olika komponenter kan utvecklas och underhållas oberoende.
3. **Innovation**: Gör det möjligt för externa utvecklare att skapa nya applikationer och funktioner ovanpå befintliga tjänster.

Fördelar med API:er

Vilka fördelar uppnår man genom att erbjuda API:er till en produkt eller tjänst?

1. **Ökad räckvidd**: Möjliggör för tredjepartsutvecklare att bygga tjänster kring produkten.
2. **Flexibilitet**: Användare kan anpassa och integrera tjänsten i sina egna system.
3. **Lägre kostnader**: Minskar utvecklingstiden genom att återanvända existerande funktionalitet.

API-design

Vad är viktigt att tänka på när man designar ett API?

1. **Enkelhet**: API:et bör vara lätt att förstå och använda.
2. **Konsistens**: Följ en enhetlig struktur och konventioner genom hela API:et.
3. **Dokumentation**: Tillhandahåll tydlig och omfattande dokumentation.

Begrepp och steg diskuteras i boken:

- Definiera resurser och deras egenskaper.
- Bestämma operationer (CRUD - Create, Read, Update, Delete).
- Säkerställa felhantering och versionering.

API-säkerhet

Vilken roll spelar säkerhet för ett API? Säkerhet är avgörande för att skydda data och säkerställa att endast auktoriserade användare kan få åtkomst till API:et. Detta inkluderar autentisering, auktorisering och kryptering.

API i ADD-processen

När kommer API (utveckling med) in i ADD-processen? API-utveckling är en del av arkitekturdesignprocessen (ADD) och bör övervägas tidigt för att säkerställa att det stödjer systemets övergripande mål och krav.

OpenAPI

Vad är OpenAPI? OpenAPI är en specifikation för att definiera RESTful API:er på ett maskinläsbart sätt, vilket möjliggör automatiserad dokumentation och klientgenerering.

Fördelar med API-teknik

Vilka fördelar ger en teknik som API ett utvecklingsprojekt och utvecklare?

1. **Snabbare utveckling**: Återanvändning av existerande tjänster minskar behovet av att bygga allt från grunden.
2. **Ökad samarbetseffektivitet**: Möjliggör samarbete mellan olika team och utvecklare.
3. **Skalbarhet**: Lättare att skala upp och utvidga funktionaliteten.

Driftsättning

Vad är driftsättning (deployment)? Driftsättning är processen att installera och konfigurera mjukvara på en server eller i en produktionsmiljö så att den blir tillgänglig för användning.

Driftsättningsbarhet

Hur definieras driftsättningsbarhet (deployability)? Driftsättningsbarhet avser hur lätt och snabbt en mjukvara kan driftsättas i en produktionsmiljö, inklusive att kunna hantera uppdateringar och förändringar.

Egenskaper som Påverkar Driftsättningsbarhet

Vilka egenskaper (angelägenheter/concerns) påverkar ett systems driftsättningsbarhet?

1. **Modularitet**: Hur väl systemet är uppdelat i separata komponenter.
2. **Automatisering**: Användning av automatiserade processer för att hantera driftsättningar.
3. **Testbarhet**: Förmågan att utföra tester innan driftsättning.

Driftsättningspipeline

Vad är en driftsättningspipeline? En driftsättningspipeline är en serie automatiserade steg som används för att bygga, testa och distribuera mjukvara. Vanliga faser inkluderar:

1. **Bygg**: Kompilera koden.
2. **Test**: Utföra automatiserade tester.
3. **Distribution**: Installera mjukvaran i en produktionsmiljö.

Continuous Integration och Deployment

Vad är continuous integration och deployment?

- **Continuous Integration (CI)**: En metod där kodändringar automatiskt byggs och testas så fort de integreras i en gemensam kodbas.
- **Continuous Deployment (CD)**: Utvidgar CI genom att automatiskt distribuera alla ändringar som passerar tester till produktionsmiljön.

Hur förhåller begreppen sig till driftsättningspipeline? Både CI och CD är centrala komponenter i en driftsättningspipeline, eftersom de möjliggör snabba och pålitliga driftsättningar.

Design av Driftsättningsarkitektur

Vad skall man tänka på när man designar driftsättningsarkitekturen?

1. **Flexibilitet**: Arkitekturen bör kunna stödja förändringar och uppdateringar.
2. **Säkerhet**: Inkludera säkerhetsåtgärder för att skydda mot obehörig åtkomst.
3. **Övervakning och loggning**: Implementera system för att övervaka och logga aktiviteter.

Designbeslut för Driftsättningsbarhet

Vilka designbeslut bidrar till att öka driftsättningsbarheten?

1. **Använda containerteknik**: Till exempel Docker, för att standardisera miljöer.
2. **Implementera mikroservice-arkitektur**: För att göra systemet mer modulärt.
3. **Automatisera tester och driftsättning**: För att minska risken för fel vid driftsättning.

Mjukvaruarkitekturens Kvalitet

På vilket sätt bidrar mjukvaruarkitekturens (den logiska arkitekturen) kvalitet till driftsättningsbarheten? En välstrukturerad mjukvaruarkitektur underlättar driftsättningar genom att separera ansvarsområden, vilket gör det lättare att uppdatera och underhålla systemet.

Återrullning ("Roll-back")

Vad innebär begreppet återrullning ("roll-back")? Återrullning refererar till processen att återställa en mjukvara till en tidigare version efter en misslyckad driftsättning eller andra problem, för att återfå systemets stabilitet.

Mönster för Driftsättningsarkitekturer

Räkna upp ett antal olika mönster för driftsättningsarkitekturer.

1. **Blue/Green Deployment**:
    - **Fördelar**: Minimalt med driftstopp, snabb återrullning.
    - **Nackdelar**: Kräver dubblering av resurser.
2. **Canary Releases**:
    - **Fördelar**: Minskar risken genom att testa nya versioner på en liten del av användarna först.
    - **Nackdelar**: Komplexitet i hanteringen av olika versioner.
3. **Rolling Updates**:
    - **Fördelar**: Gradvis uppdatering av systemet utan nedtid.
    - **Nackdelar**: Kan leda till inkonsekvenser om det inte hanteras noggrant.