Clean Architecture

Pierre Le Fevre, HI1036 VT22

Sammanfattning version 1 - 2022-02-22

Arkitektur brukar ses som längre från maskinens inre strukturer. Meningen med en bra arkitektur är att minimera mängden resurser som krävs för att driva utveckling och underhålla kodbasen. En lösning som företag med dålig kod brukar prova är att anställa fler utvecklare. Det leder dock ofta till att kostnaderna bara blir högre utan att koden förbättras, då ingen vill förstöra kodbasen med sina ändringar. I stället för att ruscha för att få ut en MVP kan arkitekturen integreras från första början. Risken är att när koden skulle förbättrats kommer en ny beställning så att originella kodbasen aldrig hinner fixas.

Mjukvara måste som namnet antyder vara mjukt, alltså lätt att ändra på. När nya funktioner behöver läggas till ska det kunna ske på ett enkelt sätt, utan att behöva redigera allt för mycket kod. Ta två program, ett som fungerar men inte går att ändra på, och ett som inte fungerar men är lätt att ändra på. Den första kommer att fungera ett tag, tills kraven förändras. Den andra kan dock modifieras för att passa kraven, och sedan redigeras när kraven förändras.

Av de tre stora programmerings-paradigmen kom *Strukturerad programmering* först. Dijkstra, som kom från matematik tyckte att bevisföring borde användas för att validera kod. Han kom dock snabbt fram till att metoden GOTO ledde till att bevisen inte kunde använda sig av mindre bevisdelar. Med if/else samt iteration kan bevisen delas upp och bevisas var för sig. När Dijkstra publicerade dessa resultat fick han dock mycket kritik av utvecklare, som tyckte att GOTO var en viktig del av programmen. Genom att koden går att se som ett matematiskt bevis kan man också använda samma filosofi med tester, alltså att bevisa motsatsen.

Objektorienterad programmeringens paradigm är en mycket mer omfattande förändring. Paradigmet är uppbyggt av tre mindre komponenter: inkapsling, arv och polymorfism. Inkapsling innebär att programmeraren kan begränsa åtkomst till olika parametrar i en datastruktur. Med arv slipper utvecklaren kopiera och klistra kod, och kan i stället bygga ut existerande klasser med det som behövs. Polymorfism tillåter dessa objekt som ärvt från en högre klass att behandlas som det. Alltså kan högre nivå-klasser interagera med objekt utan att veta om alla detaljer i objektets faktiska klass.

Funktionell programmering förlitar sig på en struktur där modifikation sker genom nästlade funktioner i stället för mutabla variabler. Om variablerna inte går att modifiera slipper systemet lida av de många problem som kan uppstå på grund av det. I stället för att förändra variablerna skulle ett system kunna implementeras där förändringarna sparas, inte det nya värdet. Varje gång det ska hämtas kan alla uppdateringar utföras och det slutgiltiga värdet räknas ut. Metoden kallas för Event sourcing.

SOLID-principerna kan användas för att strukturera upp ett mjukvarusystems mittenlager. SOLID står för: **SRP:** Single Responsibility Principle, **OCP:** Open-Closed Principle, **LSP:** Liskov Substitution Principle, **ISP:** Interface Segregation Principle, **DIP:** Dependency Inversion Principle.

SRP innebär att en komponent bara ska få ägas av en aktör. Misstag där en utvecklare på ett team råkar förändra kod som används till något som den personen inte har koll på riskerar att få oväntade resultat. Ifall kodbaserna sitter ihop blir det också svårt för de olika teamen att jobba, med tanke på

att de behöver sammanfoga deras kodbaser varje gång. I stället för att dela kod kan gemensamma bitar delas upp så att var och en bara ägs av ett team.

Med OCP menar boken att komponenter ska vara byggda för att kunna utökas, inte modifieras. Problemet med att modifiera kod är att många andra komponenter ofta beror på de delarna. Genom att i stället utöka det med nya funktioner och klasser kan andra delar av koden fortsätta som vanligt. Allt som kan göras privat i en klass bör också göra det, för att stoppa programmerare från att kringgå flödet av data som arkitekturen bestämt.

LSP förklarar att substitution bör användas i kod. Med objektorientering kan det ses som en kombination av arv och polymorfism. Genom att kunna hantera en typ av objekt som en annan typ av objekt slipper programmet omvandla och ha massa specialfall för de olika sätten som data kan vara strukturerad. Standarder bör även följas, så att alla olika submoduler fungerar på samma sätt mot huvudmodulen.

ISP förklarar att interface bör användas när olika delar av en kodbas beror på en och samma komponent. På så sätt kan utvecklingen av både underkomponenter och huvudkomponent ske utan att påverka varandra, och utan att behöva bygga om hela koden vid varje uppdatering.

DIP introducerar ett sätt att strukturera sina komponenters beroenden. Likt ISP förlitar den sig på interface, där termen Factories introduceras. Factories är abstrakta komponenter som genererar objekt av en viss typ. Genom att använda sig av dessa factories skapar utvecklaren en gräns mellan det konkreta och det abstrakta

Komponenter är mindre delar av en kodbas, i till exempel Java kan en komponent vara en JAR-fil. För att underlätta återanvändning av komponenter i äldre system behövdes Relocatability. Programmet fick först matas in till ett program (linker) som räknade ut alla relativa minnesplatser innan den kunde kompileras.

En hel del paradigmer finns i komponentvärlden också: **REP:** Reuse/release Equivalence Principle, **CCP:** Common Closure Principle, **CRP:** Common Reuse Principle. För att följa REP måste ens komponent ha versionsnummer för varje utgåva. På så sätt kan komponenter som beror på den välja en specifik version som den är kompatibel med och inte halka ur fas vid varje uppdatering. Precis som SRP säger CCP att en komponent bara ska ha en ägare. I stället ska flera olika komponenter användas så att de kan bero på den kompatibla versionen. CRP hjälper programmeraren att välja vad som ska vara med i en komponent för att balansera hur många komponenter som behövs samt hur ofta en uppdatering skapas som inte nyttjar vissa användare.

Morgonen-efter-syndrom är ett problem som upplevs i utvecklingsmiljöer där många utvecklare jobbar med samma filer. När en utvecklare skickat upp sina ändringar förstör den för de andra som jobbar, som nu behöver bygga om sina lösningar för att passa med den nya filen. För att lösa det här problemet fanns ett förslag: Alla jobbar på sina lokala kopior i 4 dagar, och den 5e spenderas på att slå ihop allt. Problemet med den här lösningen är att när projektet blir större kan det ta mer än en dag, och mängden tid spenderad på att slå ihop kod blir allt för stor.

Lösningen är att ge ut versionsnummer för de olika komponenternas utgåvor. På så sätt kan koden fortsätta köras även fast de har olika versioner på kod den beror på. När utvecklaren har tid kan den uppdatera sin kod för att anpassa sig till den nya versionen. Problemet som uppstår här är att det kan finnas cykler i beroendena som leder till att inget kan köras. För att bryta dessa cykler kan antingen ett Interface eller en ny komponent införas som båda beror på.

SDP: Stable Depencencies Principle förklarar hur ett nätverk av beroenden bör struktureras för bäst stabilitet. För att göra det lättare att kvantifiera huruvida en komponent är stabil eller inte kan mätas med hjälp av fan-in/fan-out ekvationen. Ekvationen används för att räkna antalet ingående och utgående beroenden och ger en kvot på hur stabil komponenten är. Om en viss komponent behöver vara instabil kan i stället ett Interface introduceras så att den stabila komponenter slipper bero på en instabil. För att inte koden ska bli proppfull med intefaces och abstrakta klassen, eller allt för konkret kan **SAP:** Stable Abstractions Principle användas.

Arkitekten jobbar som utvecklare i något av teamen som utvecklar produkten. Fördelarna med en bra arkitektur brukar ses efter längre tid, när systemet börjar bli stort och tyngre att utveckla. Genom att behålla en bra abstraktionsnivån kan mjukvaran till exempel byta gränssnitt och databas utan att behöva byggas om. Att splittra på koden är också viktigt, så att hela kodbasen inte blir en gigantisk monolit. Koden kan splittas i olika klasser, interface etc (Source level), i Jar/DLL-filer (Deployment level) samt i olika webbtjänster (ex. REST apier) (Service level)

Att dra gränser i arkitekturen är en viktig del av arkitektens jobb. Till exempel bör BO delen av koden inte bero på GUIt. På samma sätt bör databasen hållas separat. Men gränserna dras längre in än så. I koden lär det finnas en ”Database”-klass samt en View-klass. Dessa räknas som utanför gränsen och är inte en del av BO. Därför ska de bero på BO, och inte tvärtom.

För att bestämma vilka komponenter som ska vara längst upp kan deras *nivå* användas. Högst nivå på en komponent är den som kallas ”längst in” i koden. Den bör ligga längst upp i arkitekturen, alltså ska andra bero på den. Om man följer SRP och CCP är det dock bäst för de underliggande att kommunicera med den över Interface.

Den mest centrala delen av ett mjukvarusystem, affärslogik, kan ses som oberoende från arkitektur, programmeringsspråk, datortyp, etcetera. De reglerna hade använts utan datorer också, till exempel att räkna ut olika parametrar av ett lån inom banker. Entiteter är de som behöver lyda reglerna som specificeras i affärsreglerna. Det är ett sätt att aggregera data i strukturer där data hänger ihop för användningsområdet. De ska inte heller bero på sättet som data laddas in eller hämtas ut, utan vara helt oberoende från ex. SQL och REST.

Screaming architecture är ett begrepp som används för att beskriva hur en arkitekturs mål ska vara självklar för den som läser. Likt hur skillnaden mellan ritningar på en villa och ett bibliotek inte liknar varandra ska också kodens ritningar vara självklara. Dock ska de inte specificera på vilket sätt de tänker implementera denna ritning, och hålla sig borta från ramverk och webb i ritning och planering.

En bra arkitektur kan identifieras med följande parametrar: Fungerar oberoende från ramverk, testbar, oberoende från gränssnitt, oberoende från databas, oberoende från externa organisationen. Entiteter ska användas för att hantera data mellan affärsregler, och användningsfall ska specificeras för att bestämma hur data flödar in och ut. Utanför dessa hålls ett lager av Interface som tillåter stabil utveckling. Utanför ligger sedan olika ramverk, drivrutiner, gränssnitt, databaser, etcetera. Vid varje transiton från ett lager till ett annat bör data packas om på ett sätt som fungerar bäst för det lagret.

Humble objects är ett designmönster skapat för att splittra delarna som är svårtestade från den koden som enkelt kan enhetstestas. Till exempel kan vyn betraktas som ett *Humle object* medan en ny klass, Presenter, kan förbereda all data som vyn behövs samt testa den. Likt presentern för gränssnittet finns liknande system för databaskoppling, till exempel ORM som JPA.

I stället för att dela upp komponenter kan partiella gränser användas, där koden fortfarande ligger i samma komponent men delas av en artificiell gräns. Designmönstret Facade är ett bra exempel på detta där båda sidorna av Facaden inte kan nå den andra utan att gå igenom det menade gränssnittet.

Main-filen är sista komponenten i arkitekturen. Den ligger längst ner och inget annat i koden beror på den. Fördelen med Main-filen är att koden inte behöver vara särskilt strukturerad. Den används mest för att initiera alla komponenter som sedan startar upp resten av applikationen. Man kan använda Main-filer till sin fördel och skapa flera för att kunna köra applikationen på olika sätt, exempelvis en för produktion, en för utveckling, en för testning, etcetera.

Tjänster är ett sätt att strukturera en arkitektur som blivit väldigt populärt. På utsidan kan det se ut som att tjänsterna är helt frånkopplade, och att de är väldigt lätta att utveckla för utan att påverka de existerande versionerna. Att de är frånkopplade från varandra är till viss del sant, i och med att de körs i olika processer, kanske till och med på olika processorer. Problemet är att då data delas över nätverket måste all data skickas på samma format. Om en ände byter överföringsprotokoll måste den mottagande också ha det nya protokollet. Problemet med att uppdatera existerande tjänster är att enorma delar av koden ofta behöver redigeras för att en ny funktion ska kunna läggas till. Varje steg i flödet måste stödja nya funktionen, och dataprotokollen som skickas över nätet behöver uppdateras. System som byggs som tjänster kan oftast också byggas som komponentbaserade system. Arkitekturen behöver då anpassas enligt till exempel SOLID principerna, men in/utdata kan vara samma.

Tester bör integreras som en komponent i systemet. I stället för att gå igenom samma gränssnitt som användarna kör kan testerna i stället gå via ett eget API. Där kan programmet ge testerna full kontroll över systemet för att kunna försättas i olika lägen, och fylla variabler med förbestämda värden.

Utvecklingen av integrerade system kan dra nytta av samma principer som arkitektur av storskaliga mjukvarusystem. Det är rätt vanligt att mjukvaran som utvecklas för dessa mikrokontrollers blir bunden till en viss hårdvara, och sedan blir väldigt svårarbetad för uppdateringar och nya produkter med liknande funktionalitet men annorlunda hårdvara. Genom att dela upp kodbasen i Software, HAL, Firmware kan mjukvaran användas helt fristående från mikrokontrollern.

Om ett operativsystem används kan det vara en bra idé att abstrahera det APIet också. Ifall operativsystemet behöver bytas ut eller förändras slipper massa kodrader ändras.

I ett mjukvarusystem är databasen bara en detalj. Internt i koden refereras databasen aldrig, utan den interagerar bara med en separat adapter. Databaser har sedan länge använts för att lagra data, när det viktiga är att kunna indexera data och söka på vad som finns inuti en rad, inte till exempel namnet på en fil. Historiskt sett har databassystemen varit utformade för att kunna distribueras över hårddiskar, med deras långsamhet i eftertanke. Nu när moderna SSD/RAM-baserade system börjar dyka upp kan många i stället börja lagra sin data direkt i minne, med den datatypen som sedan kommer att användas i koden.

På samma sätt som databasen bara är en detalj av systemet kan webben ses som det. Varje gång som något händer på webb-fronten, som en ny standard eller ett nytt ramverk som alla använder förändras mycket. Det är därför bra att separera koden helt och hållet från webben. Ett bra perspektiv är att se webben som ett IO system, där mer komplexa interaktioner han hanteras av en utomstående adapter till systemet.

Ramverk bör också inte integreras i systemet, utan i stället ligga på det yttre lagret i arkitekturen, där förändringar spelar minst roll. Ett bra argument är att utvecklarna av ett ramverk ofta utvecklar det för att lösa deras problem, inte allas problem. Det är lätt hänt att ett ramverk som i en tidig version fungerat perfekt börjar halka ifrån den grundidén som den är byggd efter. Om applikationen är helt byggt utefter ramverket kan mycket av kodbasen behöva byggas om. Som boken förklarar, att använda ett ramverk är som att gifta sig asymmetriskt.

Det finns olika sätt att paketera kod. Många brukar paketera per ”lager”, till exempel MVC. Det är dock för få paket av kod när systemet börjar skalas upp ordentligt. En annan lösning är att paketera per funktion. Problemet med den metoden är att den helt plötsligt börjar gruppera kod från olika delar av applikationen som inte ens hänger ihop. Som boken förklarat kan man då använda sig av en *ports and adapters*-arkitektur. Mot mitten av arkitekturen befinner sig *domänen*, som innehåller affärsregler. Utanför den byggs flera lager där beroenden bara flödar inåt. Det kan också vara farligt att paketera per komponent då det då är lätt hänt att någon kringgår ett viktigt steg i arkitekturen i och med att de underliggande komponenterna finns tillgängliga direkt.

Reflektion

Just vårt projekt använder sig av en struktur som skulle likna microservices-arkitekturen mest. Självaste noden skulle kunna delas upp i fler delar om man följde SOLID-principerna, men som boken antyder ska man inte övertänka arkitekturen för sin lösning, utan implementera den när det är dags. Som det ser ut just nu är noden vår största tjänst och har knappast tillräckligt med kod för att en storskalig arkitektur ska vara värt det.

Precis som att det vore ineffektivt om en städare smutsade ner efter sig är det sjukt ineffektivt för en utvecklare att skapa buggar och bygga ett system som inte går att underhålla. Det bör vara ett av de centrala målen för utvecklaren att skapa ett tillförlitligt system.

SOLID principen kan jag tycker känns lite tung att implementera på den lilla skalan vi jobbar på. Även de större projekten vi gjort har inte riktigt skalan där det är värt att implementera de olika separationerna som SOLID introducerar. Datatyperna är ofta samma på de olika lagerna och val av databas och användargränssnitt är förbestämt. Dock är det mycket i SOLID som man kommit fram till själv genom att bygga mindre projekt. Saker som next-morning problemet och hur uppdelning och versionshantering är något som är centralt i hur vi utvecklar projektet just nu. Allmänt är det klart att riktigt stora system kräver ordentligt arkitektur. Det är rätt svårt att kommunicera med alla snabbt på en så stor skala, så förbestämda konkreta krav på hur data ska flöda och hur beroenden ska peka tillåter utvecklingen att ske utan att behöva stanna till och bygga om hela tiden.

Gränsdragningen hittills har känts ganska naturlig. Att dela upp den centrala logiken från webb och databas är givet. Med det pågående projektet har vi till exempel dataflöden från både HTTP och Kafka i samma Java applikation. Då är det rimligt att de får varsin del, samt att det finns en gräns mellan kommunikationssätt och logik. Just komponentbaserad utveckling på lokala maskinen, i form av olika JAR-filer är lite okänt, just delen med att bygga flera och sedan sätta ihop de. Men till exempel JavaFX använde sig av JAR-filer, så inget nytt med att importera de.