# VERTIKAL ARKITEKTUR I AGIL MILJÖ

Författare Thomas Björk, thomas.bjork@outlook.com

Datum 2024-12-17

Projektarbete inom Dataföreningen Kompetens kurs Certifierad IT-arkitekt, kurs nr 80

## Om uppsatsen

En beskrivning av hur man med vertikal arkitektur, Scrum och DevOps kan arbeta med system-utveckling i nära samarbete med intressenter på ett kostnadseffektivt sätt.

# Uppsatsen i sammandrag

Uppsatsen beskriver hur man kan arbeta med systemutveckling på ett kostnadseffektivt sätt genom att använda vertikal arkitektur, Scrum och DevOps. Det betonar vikten av att bygga applikationer i nära samarbete med användare och andra intressenter för att undvika höga kostnader och teknisk skuld som ofta uppstår med traditionella skiktade arkitekturer.

Vertikal arkitektur innebär att applikationens struktur baseras på användningsfall, där varje vertikal är separat och inte delar affärslogik med andra vertikaler. Detta tillvägagångssätt gör det möjligt att anpassa applikationen direkt efter användarens behov, vilket minskar risken för kostsamma och tidskrävande ändringar.

Dokumentet förklarar också hur vertikal arkitektur kan kombineras med agil utveckling i Scrum och en DevOps-baserad teknisk miljö för att skapa en dynamisk och effektiv utvecklingsprocess. Scrumteam arbetar i korta sprintar och har regelbundna granskningar med intressenter för att säkerställa att utvecklingen är i linje med användarens behov. DevOps möjliggör snabb återkoppling och kontinuerlig integration genom automatiserade tester och distributioner.

Sammanfattningsvis erbjuder vertikal arkitektur i kombination med agila metoder och DevOps en flexibel och effektiv lösning för systemutveckling som fokuserar på samarbete och användarens behov. [1]

[1] Sammandraget skapat av Microsoft Copilot för Word, granskat av författaren

# INNEHÅLLSFÖRTECKNING

# 1 Inledning

## 1.1 Bakgrund

Bakgrunden till den här uppsatsen är en frustration över svårigheter och kostnader inom systemutveckling och förvaltning. Många av problemen bottnar i problem med systemarkitekturen. Det är inte ovanligt att en dålig arkitektur femdubblar kostnaderna. Ett faktum jag själv råkat ut för vid flera tillfällen. Det är inte bara ett ekonomiskt problem utan skapar även en verksamhet som inte kan leva upp till omvärldens krav på snabba förändringar.

Vertikal arkitektur har diskuterats i olika sammanhang under några år. Det är ett sätt att strukturera applikationer efter användarens upplevelse. Agil metodik handlar ofta om att arbeta utgående från intressenternas krav. Vertikal arkitektur innebär att man följer den uppdelningen även i applikationens struktur.

## 1.2 Problem

Dagens mode inom systemutveckling är en skiktad arkitektur där uppdelningen i moduler styrs av tekniken som används. En vanlig uppdelning är användargränssnitt, servicelager för bearbetning, dataaccesslager för kommunikation med databasen och databas. Mellan lagren finns tydligt definierade interface. Tanken är att man genom den uppdelningen kan få en god förvaltningsbarhet genom att ett enskilt lager kan bytas ut när underliggande teknik behöver uppdateras.

I verkligheten är förändringar nästan aldrig isolerade till ett visst teknikområde i en applikation. Även små förändringar i funktionalitet kan skapa en kaskad av uppdateringar på många ställen i applikationen.

I en agil miljö är det extra viktigt att snabbt kunna hantera förändrade förutsättningar. En viktig del är en arkitektur som inte bromsar förändringar. Det vi behöver är en agil arkitektur. Förhoppningen är att en vertikal arkitektur kan vara ett svar på det behovet.

## 1.3 Syfte

Den här uppsatsen beskriver ett sätt att bygga applikationer där kopplingen mellan användarens krav och behov är direkt kopplad till strukturen i applikationen.

Grundtanken är att applikationen byggs med vertikal arkitektur där strukturen bygger på användningsfall och där varje vertikal är separat och inte delar affärslogik med andra vertikaler.

Framgångsrik systemutveckling kräver att metoder, arkitektur och programmering samverkar. För att uppnå det behöver alla delar passa ihop. Av tradition är DevOps en viktig teknik för Scrumteamet. Förhoppningen är att vertikal arkitektur kan passa in på samma sätt, att vara den mest naturliga Scrumarkitekturen.

## 1.4 Avgränsning

Vertikal arkitektur är inte en definierad metodik utan ett alternativt sätt att tänka. Det finns ingen dedikerad litteratur. Det material som finns är till stor del föreläsningar på Youtube inspelade från olika utvecklarkonferenser. Det finns ganska många om man googlar på ”Vertical slice architecture”.

Den målbild som används i den här uppsatsen är en webbaserad applikation skriven i .net med en relationsdatabas för lagring.

Vertikal arkitektur handlar om arkitekturen inom en applikation, en enskild Mikrotjänst eller liknande struktur. Relationer till omgivande system påverkas inte och diskuteras inte här.

Vertikal arkitektur fungerar i andra sammanhang men här diskuteras den i ett agilt sammanhang där Scrum är vald metodik.

## 1.5 Målgrupp

Målgruppen för uppsatsen är i första hand Scrumteam som har en uppgift att lösa. Tanken är att föra fram vertikal arkitektur som ett alternativ.

## 1.6 Metodval

Vertikal arkitektur är ingen definierad metodik och är inget hårt definierat mönster för arkitektur. Den har dykt upp som tema på föredrag på konferenser och finns framför allt på Youtube. Mönstret stämmer överens med mina reaktioner på problem inom systemutveckling och mina erfarenheter av framgångar och motgångar.

Steg ett i tekniken är att lyssna på föredrag och notera strategier och erfarenheter. Den som framför allt har diskuterat vertikal arkitektur och den som skapat begreppet från början är Jimmy Bogard, ref [BOGARD].

Steg två är eget funderande som bygger på mitt intresse för metoder, arkitektur och utveckling där jag försöker skapa en helhetsbild där Scrum och DevOps kompletterar den vertikala arkitekturen. Många tankar i den här uppsatsen bygger på egna positiva och negativa erfarenheter samlade under mer än 30 års erfarenhet av systemutveckling.

Ordförklaring

| | |
|---|---|
| Agil systemutveckling | Utveckling med fokus på korta cykler i nära samarbete med användare och andra intressenter. |
| DevOps | Miljö för automatiserad hantering av bygge och release av uppdaterade applikationer. |
| Mikrotjänst | En applikation som i kommunikation med andra applikationer löser uppgifter tillsammans. |
| Scrum | En agil utvecklingsmetodik där arbetet är uppdelat i perioder på några veckor (Sprintar). |
| SAFe | En metodik för samordning av agila utvecklingsprojekt. |
| Refaktorisering | Städning av programkod där läsbarhet och möjlighet till förvaltning förbättras utan att ändra funktionalitet. |
| Intressent | Någon som har ett intresse i projektet och kan påverka eller påverkas av dess utfall. Exempel är användare, externa projektledare och arkitekter som inte ingår i utvecklingsteamet. |
| Produktbacklog | En prioriterad lista på utvecklingsaktiviteter eller som vi säger på svenska: en TODO-lista |
| mikrotjänst-arkitektur (microservice-architecture). | Arkitektur för mikrotjänster. Brukar innebära en applikation som svarar på anrop, ofta via nätverket. Ha en privat databas. Kod som delas mellan flera mikrotjänster inom samma domän brukar läggas i en delad modul, kallad sidecar. |
| Användarberättelse (user story) | Beskriver i naturligt språk en eller flera funktioner i ett program. Fokuserar på användarens behov och nyttan av funktionen. |

| | |
|---|---|
| Användningsfall (use case) | En beskrivning av hur användaren interagerar med systemet som är mer detaljerad än en användarberättelse. |

# 2 Problemanalys

Dagens systemutveckling har många problem. Problembilden skiljer sig mellan olika organisationer men många är allmängiltiga. Det handlar om höga kostnader, långa utvecklingstider och ett slutresultat som inte alltid når upp till önskad nivå. Med tanke på hur många intelligenta människor som är inblandade kan man undra över varför det är så svårt. Ref [BARR].

## 2.1 Agil systemutveckling

Agil systemutveckling har varit aktuell länge. Tanken är att arbeta inkrementellt med stort fokus på användare och andra intressenters behov. Snabb återkoppling och flexibilitet genomsyrar arbetet. Förändringar i kraven välkomnas och en levande dialog mellan team och intressenter är av högsta prioritet.

Den agila miljön ställer speciella krav på arkitekturen. För att hålla dialogen levande med intressenter är det viktigt att regelbundet visa upp resultatet av arbetet. Genom att visa upp funktionalitet kan man skapa intresse på ett helt annat sätt än genom dokument eller powerpointpresentationer. Eller som det står i agila manifestet ”Vår högsta prioritet är att tillfredsställa kunden genom tidig och kontinuerlig leverans av värdefull programvara.”, ref [AGIL].

En viktig punkt i agil systemutveckling är engagerade intressenter. Genom att ha en levande dialog med användare skapar man gemensamt en vision om vad applikationen skall göra. Om utvecklingen är långsam och teknikfokuserad finns det ofta inget att visa som intressenterna är intresserade av. Då är det svårt att hålla dialogen levande.

Arkitekturen måste även hantera förändringar på ett bra sätt utan att påverka andra delar. När en förändring införs skall den påverka andra delar så lite som möjligt.

I en arkitektur försöker man alltid sträva efter låg koppling mellan komponenterna och hög sammanhållning i komponenten (Coupling and Cohesion). Klassisk arkitektur delar upp en applikation efter vilken teknik som används. Det fungerar illa i en agil miljö. Vi behöver en arkitektur som definierar komponenter på ett sätt som matchar användarnas syn på applikationen.

## 2.2 Arkitekter

Det finns ibland en konflikt mellan arkitekter och agil verksamhet. Arkitekter vill ha stabila förutsättningar, ordnat beslutsfattande och lägger stor vikt på dokumentation. Agil verksamhet vill ha flexibla förutsättningar, decentraliserat beslutsfattande och föredrar fungerande programvara framför dokumentation.

I ett större sammanhang är arkitektens syn den viktigaste. Verksamhet, driftsmiljöer och samverkan mellan applikationer är inget som ändras i en sprint. Ett exempel är skillnaden mellan SAFe och Scrum som ofta används parallellt. SAFe är ordnad med tidshorisonter och planer. Scrum jobbar med kort tidshorisont och är hela tiden öppen för förändringar.

Det är viktigt att arkitekten förstår och kan hantera skillnaden i olika sammanhang. I Scrumsammanhang är det viktigt att arkitekten kan hjälpa till med en arkitektur som är flexibel och underlättar den agila processen.

## 2.3 Kostnad

Utveckling och förvaltning av mjukvara kostar enorma belopp varje år och kostnaden ökar. Samhällets beroende av datorbaserade lösningar accelererar. Tyvärr ökar produktiviteten inte i samma omfattning så skillnaden mellan behov och tillgängliga resurser ökar.

Det är förvånansvärt ofta man inte tänker i termer av kostnader. Den betraktas som en konstant som inte kan påverkas. Med det sättet att tänka lyfter man inte heller problem och letar inte efter lösningar.

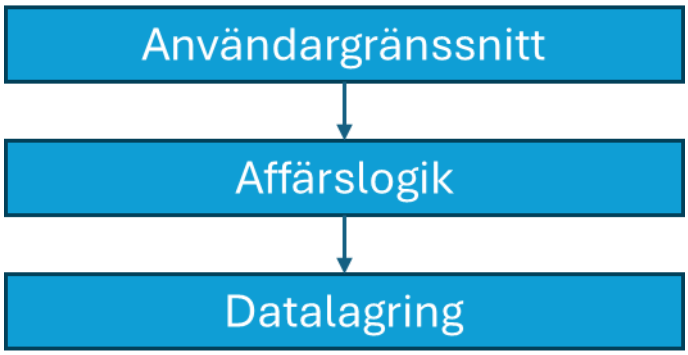
Arkitektur är kanske den mest kostnadsdrivande aktiviteten inom systemutveckling. Både positivt och negativt. Det innebär även att det bästa sättet att kapa kostnaderna och skapa en effektiv verksamhet är att fokusera på arkitektur. Det räcker inte med att anställa arkitekter som träffar teamet några gånger per år. Kunskapen om arkitektur måste finnas inom teamet och hela tiden utvecklas.

Min ”finger-i-luften”-utvärdering av vad dålig arkitektur kostar säger att det ofta handlar om 5-dubblad kostnadsökning. I en stor organisation märks inte ens det problemet. Man vet inte vad utvecklingen kan kosta utan tillsätter resurser utan att utvärdera hur man kan arbetar effektivare.

## 2.4 Skiktad arkitektur

Skiktad arkitektur går under många namn och det finns olika tolkningar av den men en referens är ”Clean architecture” av Uncle Bob (Robert C. Martin), ref [BOB 01]. Strukturen är så allmänt accepterad att den är norm, något man väljer mer eller mindre med automatik.

Den typen av arkitektur handlar om att skapa en skiktning mellan tekniklager. Grundmodellen delar upp i användargränssnitt, affärslogik och datalagring. Kommunikation inom varje skikt är i princip fri men varje skikt är frikopplad från övriga och kommunicerar enbart indirekt genom interface. Genom den uppdelningen kan varje skikt testas separat och vid teknikbyte, till exempel byte av databas, kan ett skikt (teoretiskt i alla fall) bytas ut utan att övriga skikt påverkas.




**FIGUR 1. SKIKTAD ARKITEKTUR**

Den skiktade arkitekturen är i teorin prydlig och välstrukturerad. I praktiken är den problematisk på många sätt. Felaktigt utförd är den tungarbetad och skapar snabbt teknisk skuld.

Ett problem är att modellen frikopplar utvecklingen från användarens upplevelser. Varje interaktion från användaren påverkar flera, kanske alla, lager i systemet. Varje nytt eller uppdaterat användningsfall skapar en kaskad av uppdateringar i applikationen. Arbetsgången gör att det inte blir ett jämnt flöde av levererad funktionalitet och man lätt hamnar i ett läge där stora delen av applikationen utvecklas utan återkoppling från användare.

Testning blir ett problem. Uppdelningen i skikt gör det svårt att skapa automatiserade testfall som utgår från användningsfall. I stället testar man funktioner i respektive lager. Varje lager testas separat och övriga lager simuleras. Det är en tidskrävande aktivitet. Testerna är inte särskilt effektiva, svåra att förstå och svåra att förvalta.

Ursprunget till arkitekturen är arbetsmodellen där separata team implementerar olika delar enligt i förväg uppgjorda specifikationer och mot slutet integrerar modulerna till en enhet och därefter lämnar över till testning.  Den uppdelningen är mindre vanlig när man lämnar vattenfallsmodellen och arbetar med agila team. Organisationen driver inte längre arkitekturen men modellen lever kvar.

Men även om alla skikten byggs av samma team blir uppdelningen problematisk genom att den inte följer strukturen i aktiviteterna. Det blir lätt att Scrumteamet lägger in egna aktiviteter för att hantera skiktningen och tappar närheten till intressenter. En agil miljö behöver en agil arkitektur och då är den skiktade modellen inte optimal.

En betydligt mer genomgående analys av för- och nackdelar med skiktad arkitektur finns i kapitel 10 av ref [FORD_1]. Där beskrivs den som enkel att komma i gång med men har problem när applikationen växer.

# 3 Vertikal arkitektur och agil utveckling

## 3.1 En sammanhållen berättelse

Att vara agil fullt ut kräver att metoder, teknik och arkitektur skapar en helhet. Genom att alla delar samverkar minskar friktionen och ger förutsättning att koncentrera på leveransen. Intressenternas berättelse passar in och ger ett naturligt flyt.

Grunden i den här uppsatsen är att utforska möjligheterna med en vertikal arkitektur. Det är en enkel teknik som kan vara intressant i sitt eget sammanhang. Men det som gör den verkligt intressant är hur den tillsammans med agil utvecklingsmetodik kan skapa en dynamisk och effektiv systemutveckling och förvaltning med fokus på samarbete.

Målet är att skapa systemet som en berättelse där intressenter och utvecklingsteam har en gemensam vision om vad som skall åstadkommas. Det uppnås genom att hela tiden ha en levande dialog. Fokusera på det som driver berättelsen framåt. Var synlig och intressant.

Modellen är en miljö där utvecklingsprocessen är Scrum, tekniska miljön är DevOps-baserad och utveckling enligt ”Vertikal slice architecture”. Effektiv systemutveckling handlar om att få teknik, metoder och organisation att fungera som en enhet.

Genom att arbeta med vertikal arkitektur, Scrum och DevOps får man ett ramverk för berättelsen. All verksamhet behöver hela tiden arbeta med återkoppling och förbättringar.

## 3.2 Vertikal arkitektur

Att bygga en applikation med vertikal arkitektur innebär att man delar upp efter användarberättelser (user stories). Det som användaren ser som en naturlig uppdelning är det som styr var gränserna sätts. Varje vertikal är självständig och anropar aldrig direkt kod i andra vertikaler.

## 3.3 Scrum

Scrum är en agil metodik som används för olika typer av utvecklingsprojekt men är framför allt använd inom utveckling av programvara. Genom att utveckla i nära samarbete med användare och andra intressenter kan teamet fokusera på att skapa mervärde. Snabb återkoppling ger möjlighet att justera funktionalitet på tidigt stadium och skapa produkter som motsvarar behovet.

I Scrum arbetar man i korta cykler (Sprintar) på 1 till 4 veckor. Efter varje sprint träffas team och intressenter och utvärderar vad som åstadkommits och diskuterar vad som skall vara med i nästa sprint.

Hjärtat i utvecklingen är en lista på önskemål (Produktbacklogg). Listan är prioriterad och teamet plockar in ett lämpligt antal aktiviteter inför varje sprint.

I en vertikal arkitektur fokuseras utvecklingen på synlig funktionalitet och att snabbt komma i mål med enskilda aktiviteter. Exakt samma fokus som gäller inom Scrum och implementation av posterna i backloggen. Genom att jobba vertikalt får man en rak koppling mellan intressenternas behov och vad som utvecklas.

Scrum och vertikal arkitektur passar som hand i handske och kan vara nyckeln till framgångsrik agil utveckling.

Scrum beskrivs i Scrum-guiden och finns även i svensk översättning. Se referens [Scrum].

Som bilaga till den här uppsatsen finns en genomgång av hur Scrum kan fungera tillsammans med vertikal arkitektur. Det är ett komplement till Scrum-guiden och följer i stora drag strukturen på Scrum-guiden.

## 3.4 DevOps

En förkortning av development och operations. Innebär att utvecklingsteam och driftteam samarbetar i en automatiserad miljö för bygge, test och produktionssättning av applikationer. Ett DevOps-flöde kan ha följande steg.

1) Utvecklaren är klar med en funktion eller rättning av programvaran
2) Kodändringen checkas in i systemet för versionshantering
3) En byggserver kompilerar applikationen och kör automatiska tester
4) Den nya versionen finns tillgänglig internt
5) En produktägare bestämmer om versionen skall distribueras externt.
6) Om ja, distribueras nya versionen externt

Alla steg utom 1 och 5 är automatiserade. En fungerande DevOps-miljö ger snabb återkoppling till team och intressenter genom att aktuell version alltid finns tillgänglig, är testad och kan utvärderas. Det underlättar samarbetet mellan utvecklarteam och intressenter.

# 4 Vertikal arkitektur

## 4.1 Översikt

Grunden för vertikal arkitektur är enkel.

*Det som hör ihop skall sitta ihop*
*Det som inte hör ihop skall inte sitta ihop*

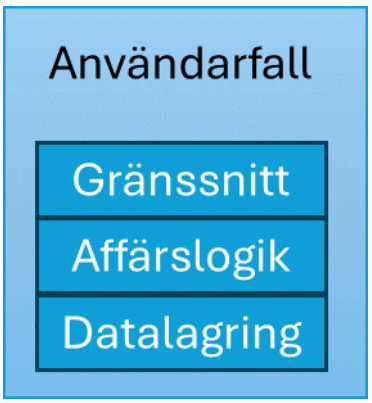
Den principen för arkitektur är allmängiltig och gäller i alla sammanhang. Det som skiljer sig är definitionen av vad som skall sitta ihop. Det är inte längre teknik som definierar vad som hör ihop utan funktion. Mjukvara lever i en virtuell värld med andra fysiska begränsningar. Det skapar möjlighet att även omdefiniera grundläggande koncept.

I en vertikal arkitektur grupperas kod efter vad som implementeras och utgår från användarens upplevelser. I en användarintensiv applikation kan en vertikal bestå av flera användningsfall som är kopplade till samma berättelse.

Vertikal arkitektur ger utvecklarteamet större flexibilitet och möjlighet att jobba effektivt i nära samarbete med intressenter. Men det innebär även ett större ansvar att göra ett bra jobb. Strukturen är inte fast utan måste hela tiden utvärderas. Skriven kod testas, städas, granskas och utvärderas.

## 4.2 Användningsfall implementeras i separata vertikaler

Uppdelning i vertikaler skall ha en stark koppling till användningsfall. Ofta är kopplingen ganska rak men det är en rekommendation och inte en regel. Uppdelningen måste göras genomtänkt. I en applikation kan det innebära en användarberättelse (user story) ger en vertikal med fler än en funktion.




**FIGUR 2. STRUKTUR FÖR EN VERTIKAL**

Ett exempel. I en webb-applikation kan en sida eller delsida implementeras i en vertikal. I en redigeringsbild, till exempel för adresser, kan det innebära att alla redigeringsfunktioner (Läsa, redigera, radera) hanteras av samma vertikal.

En bra guide kan vara att tänka som en användare och hitta användarens logiska sammanhang.

## 4.3 Vertikaler ansvarar för sin egen struktur

En vertikal består av hårt kopplad kod vars enda uppgift är att lösa användarens behov. Skiktning och struktur är upp till utvecklaren. Att jobba i vertikaler är enklare på så sätt att man inte

behöver lägga tid  på onödig och överarbetad uppdelning av kod. Det gör koden sammanhållen och fokuserad på uppgiften.

Däremot är ansvaret större att utvecklaren skriver kod relevant för uppgiften. Man kan inte längre följa ett fast mönster utan måste hela tiden arbeta med testbarhet och struktur. Vertikal arkitektur är frihet under ansvar.

Refaktorisering av kod innebär att koden under utvecklingsfasen modifieras i flera steg för att skapa läsbarhet och förvaltningsbarhet. Att aktivt arbeta med refaktorisering i utveckling med vertikal arkitektur är absolut nödvändigt.

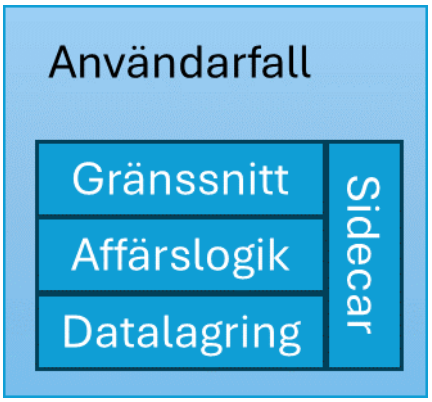
Rekommenderad referens: ”Refactoring: Improving the Design of Existing Code” av Martin Fowler, ref [FOW01]

Det är även viktigt att arbeta med kodgranskningar inom teamet för att inte olika stilar skall skapa för stora skillnader mellan likartade vertikaler.

Ingången till en vertikal är ett användningsfall . Vertikaler ligger parallellt och anropar inte varandra. Om det finns behov att koppla ihop vertikaler görs det till exempel via asynkron meddelandeförmedling.

## 4.4 Sidecar

Sidecar är ett begrepp som ofta används inom mikrotjänst-arkitektur (microservice-architecture). Det är en samling moduler som löser uppgifter som är gemensamma för flera, kanske alla, mikrotjänster. Framför allt är det i den miljön viktigt att ha en gemensam hantering av kommunikation och säkerhet.




**FIGUR 3. VERTIKAL MED SIDECAR**

I en vertikal arkitektur finns stora vinster med att dela gemensamma funktioner som inte är affärslogik. För att jobba effektivt är bra hjälprutiner viktigt. Det avlastar utvecklaren om det finns grundläggande regler för standardfunktioner.

Exempel på funktionalitet i sidecar-modulen kan vara

- Loggning
- Autentisering
- Behörighet
- Konfiguration
- Felhantering

Det är viktigt att man tidigt i utvecklingen samlar funktioner som underlättar utvecklingen. Ofta handlar det om att installera rätt paket med färdig kod, konfigurera och definiera hur användningen skall gå till. Bara i undantagsfall är det funktioner som implementeras lokalt.

## 4.5 Delad affärslogik

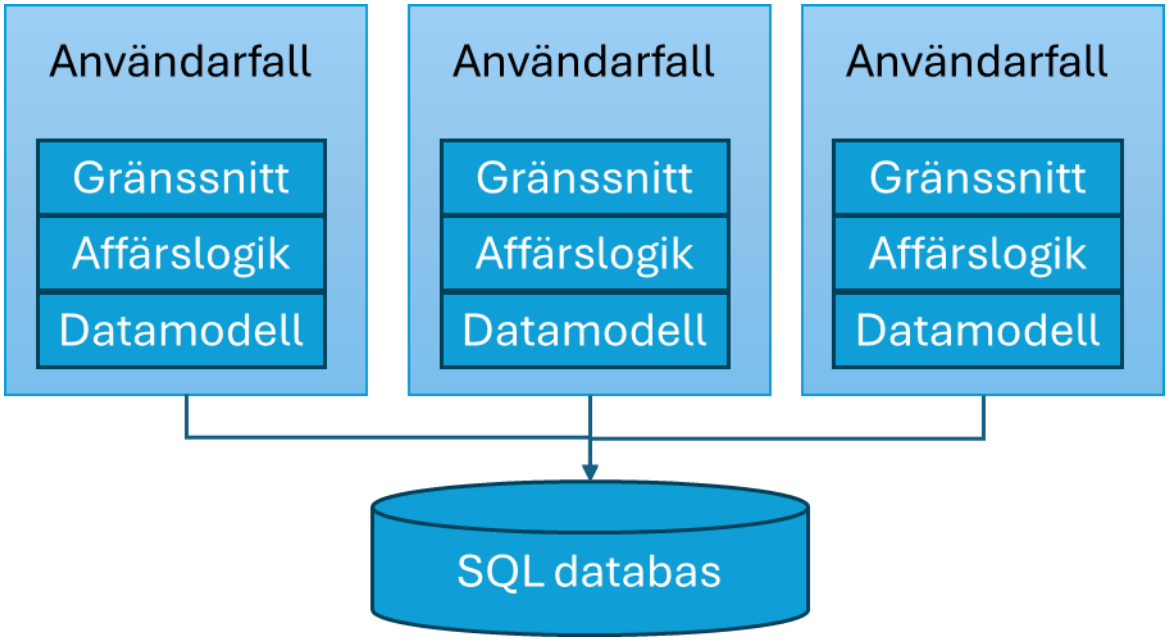
Det är inte förbjudet att ha en sidecar med affärslogik men det skall göras med försiktighet och bara om det finns en verklig nytta. Ofta blir delad kod mer komplicerad när flera moduler använder den eftersom behovet kan skilja sig mellan olika användningsfall. Kostnader är sällan kopplade till kodvolymen utan till komplexitet.

Samtidigt är det risker och kostnader kopplade till att kopiera samma funktionalitet till flera ställen. Det skapar problem med samordning när en ändring påverkar flera platser. Avvägningen mellan delad och kopierad funktionalitet måste hela tiden utvärderas. Ofta skapas funktionaliteten på ett ställe i applikationen men när den används i tre eller fler vertikaler kan det vara lämpligt att bryta loss den och lägga i en sidecar.

## 4.6 Delad databas

I en vertikal arkitektur vill man ha separation mellan vertikaler så långt möjligt. Det ställer till problem när applikationen har en delad databas. Att varje vertikal har sin egen databas eller sina egna tabeller i databasen är sällan ett alternativ. Vi behöver en modell där vertikaler kan samverka mot samma databas men arbeta separat.

En lösning är att använda ett ORM-system som till exempel Entity Framework eller NHibernate. ORM betyder Object-Relational Mapper och innebär att man skapar klasser i applikationen där varje tabell i databasen motsvaras av en klass. Fälten i databasen får motsvarande attribut i klassen. Rader i databasen hanteras genom att objekt skapas. En rad i databasen get ett objekt. Redigering av databasen sker genom att ändringar utförs på objekten som därefter synkroniseras med databasen. En längre introduktion finns i ref [ORM].




**FIGUR 4. VERTIKALER MED DELAD DATABAS**

Eftersom vi vill ha en så separerad lösning som möjligt men en gemensam databas skapar man unika klasser i de vertikaler som är intresserade av motsvarande tabeller. Om det finns en

persontabell i databasen och två vertikaler vill jobba med personer skapar man egna Person-klasser i respektive vertikal. Den ena vertikalen kanske bara intresserad att läsa namnet och där finns bara attribut för att läsa namnet. Den andra vertikalen hanterar adressändringar och behöver läsa, uppdatera och skapa personer i databasen. I den vertikalen finns ett personobjekt med både läs- och skrivrättigheter.

Genom att varje vertikal har sina egna ORM-klasser är det även lämpligt att lägga in programlogik som hör till datatypen. Ett exempel är formatering av namn. Förnamn och efternamn kanske ligger i separata fält i databasen men vertikalen vill ha förnamn och efternamn tillsammans. Den hanteringen kan läggas som ett beräknat fält i ORM-objektet.

En annan funktion som kan vara intressant är fältvalidering. Kontroll av obligatoriska fält, till exempel Namn, är ofta ett krav i ett personregister och det finns regler för hur personnummer skall vara uppbyggda. Genom att trycka ner den typen av kontroller i ORM-objektet blir programlogiken som jobbar med personer enklare.

Säkerheten i systemet blir högre om varje vertikal bara har tillgång till det som behövs och genom att bara ha skrivrättigheter till delar av databasen som hanteras av vertikalen.

Databasen är ofta basen i en applikation och det är viktigt att ha ett samlat grepp om databasstrukturen. Ändringar i databasen är ofta komplicerade, speciellt om grundstrukturen påverkas av ändringen. Refaktorisering av en databas är svårt och skall undvikas i det längsta. Även om varje vertikal jobbar oberoende av varandra är det viktigt att grundstrukturen blir rätt från början. Om det lyckas kan databasen växa fram över tid.

För att tillägg till databasen inte skall ställa till problem för andra vertikaler är det några regler att tänka på.

- Skapa en väldefinierad grundstruktur tidigt i utvecklingen.
- Ha regler för namnsättning och andra attribut på databasen.
- Undvik constraints i databasen. Constraints är regler som implementeras i databasen för att strukturen på data alltid skall vara konsistent. I en central databas som används av flera applikationer är det nödvändigt. I en databas som ägs av en specifik applikation skapar begränsningarna mest bekymmer. Testfall är ett bättre sätt att verifiera att applikationen lagrar data på ett bra sätt.
- Ändringar skall påverka så lite som möjligt. Nya fält skall ha defaultvärde eller tillåta odefinierade värden (null). På så sätt behöver inte alla vertikaler uppdateras när en enskild vertikal behöver ett tillägg.
- Testning är viktig. Varje vertikal måste ha testfall för sin egen användning av databasen. Efter uppdatering av databasen skall tester för alla vertikaler köras.
- Testning görs mot en riktig databas. Mockning eller annan typ av simulerad databas hittar sällan problem orsakade av databasförändringar.
- Om databasen behöver ändras på ett sätt så flera vertikaler påverkas hanteras det som en separat aktivitet i backlogen.

## 4.7 Dependency injection

Dependency injection innebär att kopplingen mellan moduler skapas när applikationen körs. Varje modul känner bara till en definition av vilken funktionalitet som är tillgänglig i andra moduler men inte vad som implementerar funktionaliteten.

Tekniken implementeras ofta genom att det vid uppstart av applikationen körs en rutin som kopplar interface till en specifik implementation. Det gör att man till exempel vid testning kan byta ut vissa moduler mot simuleringar. Praktiskt när man testar en modul som skickar epost och inte vill skicka tusentals mejl till hela världen.

Rätt använd är tekniken värdefull men inte utan problem. Genom att beroende byggs dynamiskt när applikationen körs skapar det en svåröverskådlig applikation för utvecklaren. Det går inte på ett enkelt sätt att läsa koden och veta vad som anropas.

I en vertikal arkitektur är kopplingen starkare inom vertikalen. Då är behovet av lösa dynamiska kopplingar betydligt mindre. Undvik därför dependency injection där det inte skapar mervärde eller löser problem.

## 4.8 Testdriven utveckling

Inom en vertikal är kopplingar mellan delar inom vertikalen stark. Det innebär att det inte finns någon mall för uppdelning och skiktning. Det som styr är programlogik och logisk struktur. Det finns inga krav att dela upp efter teknik.

Däremot finns krav på att koden fungerar som tänkt. I Scrum finns ingen separat testfas utan avslutad aktivitet förväntas uppfylla intressentens förväntning. För att uppnå det är det nödvändigt att utvecklaren förstår hur man skriver automatiserade tester samtidigt som användningsfallet implementeras.

En teknik som ofta rekommenderas i sammanhanget är Testdriven utveckling, TDD. I en vertikal arkitektur är det extra viktigt att ha bra automatiserade tester eftersom det är utvecklarens ansvar att skapa en bra intern kodstruktur och med TDD kan struktur förbättras på ett säkert och effektivt sätt. Ref [TDD].

Arbetsgången i TDD är lite förkortat enligt nedan

1. Utgå från användningsfallet som skall implementeras
    - Skriv ett test som testar användningsfallet eller en del av användningsfallet.
    - Testet skall beskriva förväntat beteende för funktionen.
2. Kör testet
    - Kör det nyskrivna testet för att säkerställa att det misslyckas.
    - Detta steg verifierar att testet är korrekt och att funktionen inte redan existerar.
3. Implementera koden
    - Skriv den minsta möjliga mängd kod som krävs för att få testet att passera. Fokusera på att få testet att gå igenom, inte på att skriva perfekt kod.
    - Kör alla enhetstester för att säkerställa att den nya koden inte bryter befintlig funktionalitet. Om något test misslyckas, justera koden tills alla tester går igenom.
4. Refaktorisera koden
    - Förbättra koden utan att ändra dess funktionalitet. Se till att koden är ren, effektiv och lätt att underhålla.
    - Kör alla tester igen för att säkerställa att refaktoriseringen inte har introducerat några fel.
5. Upprepa processen
    - Fortsätt med nästa funktion eller förbättring och upprepa stegen ovan.

En stor fördel med vertikal arkitektur är att tester är begränsade till aktuell vertikal. Tester mot andra vertikaler behöver inte köras så länge förändringen är lokal.

# 5 Vertikal arkitektur i befintliga system

Utveckling i projektform är på tillbakagång. Numera är systemutveckling ofta hanterad av produktteam som arbetar inom ett affärsområde eller med en specifik produkt under lång tid. Även i ett renodlat utvecklingsprojekt är nästan all utveckling förändringar i befintliga system. I en Scrummiljö är det bara början på första sprinten som är ren nyutveckling. Resten är förvaltning och vidareutveckling.

Det här kapitlet ger lite tips på hur ett befintligt system kan ha nytta av ett vertikalt tänkande.

## 5.1 Webb-applikation

En webb-applikation som är uppdelad i separata sidor eller delsidor kan göras mer vertikal. Genom att gruppera sidorna som separata vertikaler kan programlogiken bli mer separerad och vidareutvecklas med färre konflikter.

## 5.2 Testning

Ett testfallsbibliotek som är strukturerat kring användningsfall är mer användbart och lättare att förvalta. Tester blir stabilare genom att förändringar i intern funktionalitet inte påverkar tester så länge funktionalitet inte förändras. Det är bara vid funktionsförändringar som testfall påverkas.

Att skriva testfall som är stabila över tiden är målet. Med en vertikal arkitektur där testerna följer användningsfallen och testerna grupperas tillsammans och äger sina egna interna beroenden skapas en naturlig struktur som underlättar både vidareutveckling och felsökning.

Vertikal struktur på tester kan användas även i applikationer som inte är byggda med vertikal arkitektur.

## 5.3 Refaktorisering av monolit

Det här är ett extremfall som behöver tänkas igenom noga. En illa genomförd refaktorisering kan skapa ett system i betydligt sämre skick än utgångsläget. Riskerna kan minimeras genom att arbeta inkrementellt. Principen är att man grupperar koden i vertikaler och därefter refaktoriserar varje del separat. Tekniken bygger på verktyg i utvecklingsmiljön där det finns automatiserat stöd för att ändra struktur och flytta runt kod utan att skada applikationen.

1. Gruppera koden i lämpliga vertikaler. All kod som används inom en vertikal kopieras. Det kan innebära att stora mängder kod dubbleras.
2. Skapa automatiska tester mot respektive vertikal. Finns det tester mot monoliten delas även dessa upp. Testerna skall vara heltäckande och av hög kvalitet.
3. Ta bort onödiga interface inom respektive vertikal och ändra anrop att gå direkt. Det måste vara extremt tydligt för utvecklingsmiljön och utvecklaren vilken kod som anropas vid varje tillfälle. Interface skapar en osäkerhet genom att beroendet byggs dynamiskt när applikationen körs.
4. Radera död kod, dvs kod som inte kan anropas.
5. Refaktorisera varje vertikal. Genom att koden nu bara tjänar sin egen vertikal finns stora möjligheter till förenkling och uppstädning.

För den som vill bryta isär en monolit rekommenderas boken ”Software Architecture – the hard parts”, ref [FORD_2], som innehåller flera kapitel i ämnet.

## 5.4 Utbrytning av mikrotjänst

Vertikal arkitektur kan ses som en lättversion av mikrotjänst-arkitektur. Förenklingen är att vertikalerna ingår i samma applikation och man har en delad databas. Det kan samtidigt begränsa möjligheten att skala applikationen och även uppdelning av utvecklingen i separata team.

Om det orsakar problem kan en uppdelning av applikationen i flera mikrotjänster vara en lösning. Fördelen med den vertikala arkitekturen är att den har få beroenden och uppdelningen i separata delar som körs separat är enklare än för monolit-arkitekturer. Det som behöver hanteras är framför allt hanteringen av databasen. Varje mikrotjänst har sin egen databas medan vertikalerna har en delad.

# 6 Slutsatser och rekommendationer

Vertikal Arkitektur tillsammans med Scrum och DevOps skapar en sammanhållen berättelse om systemutveckling. Under rätt förutsättning kan det ge kostnadseffektiv utveckling av system som ger nöjda intressenter.

Konceptet är så enkelt att det kan sammanfattas med några få meningar men när det genomsyrar hela utvecklingskedjan är det många detaljer som skall hanteras. I varje sammanhang är det viktigt att aktivt leta efter det bästa sättet att utföra uppgiften. Vertikal arkitektur är ett inlägg i den debatten och kan vara ett viktigt bidrag men är inte alltid den bästa lösningen.

Vertikal arkitektur är en princip och inte en metodik. Varje tillämpning måste själv definiera vad det innebär och hur det passar in i den specifika verksamheten. Tekniken är lovande i rätt sammanhang men det finns problem med ett införande.

Det största problemet handlar om att det är ett koncept som bryter det som anses vara den enda rätta vägen. Skiktning beroende på teknik är så allmänt förekommande att det anses vara ett av fundamenten i systemutveckling.

Ett annat problem är att en vertikal arkitektur ställer högre krav på gruppen utvecklare. Inom en vertikal är strukturen relativt fri, det är utvecklarens ansvar att skriva fungerande förvaltningsbar kod. Det är inte alla utvecklare som ligger på rätt kompetensnivå för att ta det ansvaret.

Att utveckla i en miljö med snabb återkoppling och engagerade intressenter är systemutvecklingens heliga gral. Kan man uppnå det skapar det en fokus bland team och intressenter som är svårslaget.

# 7 Diskussion

Att definiera ett koncept som är så odefinierat som Vertikal Arkitektur är en utmaning. Konceptet är enkelt, på gränsen till trivialt. Utvecklingen går tillbaka till grundläggande programmering med enkla funktioner som löser enkla problem i ett enkelt sammanhang.

Samtidigt har systemutveckling och tillhörande arkitektur enorma utmaningar. På många sätt har utvecklingen stått still eller gått bakåt när det gäller framför allt produktivitet. De flesta applikationer är betydligt större än i äldre tider. Det är inte ovanligt att skiktningen gått helt över styr. Automatiserad testning anses ofta vara något exotiskt bland utvecklare. Kontakt med intressenter är något nödvändigt ont som Scrum mastern hanterar. Produktägare har ingen djup kunskap om applikationen utan ägnar sin tid åt att sitta i möten och jobbar med releaseplaner tillsammans med ledningsgruppen eller sitter fast i sin egen erfarenhet av hur det var förr. Arkitekten har en gång i tiden skapat en plan eller prototyp och därefter försvunnit. Hen dyker upp någon gång i kvartalet och svaret på alla förslag till förbättringar är alltid ”Nej, vi gör som vi har bestämt tidigare.”

I en sådan miljö är det svårt att få till förbättringar. När ett agilt arbetssätt skall införas brukar man anlita en agil coach som skall leda och inspirera till förändringen. Kanske är det dags att arkitekterna kommer ner från elfenbenstornet och tar ledarskapet i en förändring mot en mer genomtänkt arkitektur som är i samklang med teknik och verksamhet.

# 8 Referenser

[BOGARD] Jimmy Bogard, 2018, ”Vertical Slice Architecture”, https://www.jimmybogard.com/vertical-slice-architecture/

[FOW01] Martin Fowler, 2019, “Refactoring: Improving the Design of Existing Code”. Second Edition. Addison-Wesley Professional ISBN 978-0-13-475759-9.

[FORD_1] Mark Richards, Neal Ford, 2020, ”Fundamentals of Software Architecture”. O’Reilly Media, Inc. ISBN: 9 781 492 043 454

[FORD_2] Neal Ford, Mark Richards, Pramod Sadalage, Zhamak Dehghani, 2022, “Software Architecture: The Hard Parts”. O’Reilly Media, Inc. ISBN: 978-1-492-08689-5

[BOB 01] Robert C. Martin, 2018, “Clean Architecture, A Craftsman's Guide to Software Structure and Design”, Pearson Education, Inc. ISBN 978-0-13-449416-6

[BARR] Adam Barr,2018, “The Problem with software, why smart engineers write bad code”, MIT Press, ISBN 978-0-262-03851-5

[JACOB] Ivar Jacobson, “Use Cases – Reframed for Modern Product Development”, https://www.ivarjacobson.com/software-development-engineering/use-cases

[SCRUM] Ken Schwaber and Jeff Sutherland, ”The 2020 Scrum Guide” https://scrumguides.org/

[ORM] Mia Liang, ” Understanding Object-Relational Mapping: Pros, Cons, and Types” https://www.altexsoft.com/blog/object-relational-mapping/

[TDD] Kent Beck, 2002, ”Test Driven Development: By Example”, Addison-Wesley Professional, ISBN 978-0-321-14653-3

[AGIL] “Manifest för Agil systemutveckling”, https://agilemanifesto.org/iso/sv/manifesto.html

Microsoft Copilot har använts som bollplank i arbetet och hjälp med formuleringar men åsikter och slutsatser kommer från ovanstående referenser men framför allt författarens egna erfarenheter och synpunkter.

# 1 Bilaga. Scrum och vertikal arkitektur

## 1.1 Introduktion

En vertikal arkitektur ger möjligheter till korta utvecklingscykler med tydlig koppling mellan utveckling och levererad funktionalitet. Den miljön är passar utmärkt för agil utveckling enligt Scrum.

Scrum är ett agilt ramverk med fokus på nära samarbete med intressenter, synlighet och snabb återkoppling. Den starka kopplingen mellan implementation och intressenternas berättelser passar perfekt med arbetssättet i Scrum.

Scrum är en generell metod för agil utveckling och är Scrumguiden är beskriven på en generell nivå. Den här bilagan är kommentarer och tillägg som kompletterar Scrumguiden när man arbetar med vertikal arkitektur och interagerar med intressenter.  Det är inte på något sätt en ersättning till Scrumguiden. Att arbeta effektivt med Scrum kräver full förståelse för den kompletta guiden.

Läs den här bilagan som ett komplement till Scrum-guiden, kanske parallellt. Strukturen följer Scrumguiden och citat från Scrumguiden finns inlagda. Dessa är markerade med *kursiv* stil.

Scrum-guidens referens, se [SCRUM].

## 1.2 Scrum arbetsflöde

Scrum är en iterativ metod. Varje aktivitet ingår i ett tidsfönster på 1–4 veckor (sprint). Utgångspunkten för arbetet är en todo-lista med aktiviteter (product backlog). I början på varje sprint väljer teamet ut ett antal aktiviteter som skall implementeras under sprinten. Varje sprint har ett formulerat mål (Sprintmål, Sprint Goal) som sammanfattar vad som skall implementeras.

Varje dag håller teamet ett kort möte. Man följer upp hur arbetet fortskrider mot målet och gör justeringar vid behov.

Det finns en definition av vad som skall vara uppfyllt för att en aktivitet skall vara avslutad (definition av klar, definition of done). När en aktivitet uppfyller kraven anses den vara avklarad.

Resultatet av sprinten är en uppdaterad version (ett inkrement) av den levererade produkten.

I slutet av sprinten hålls ett möte mellan team och intressenter där resultatet utvärderas. (Sprintgranskning, Sprint Review)

I samband med sprintslut håller även teamet ett möte för utvärdering och planering för ökad kvalitet och effektivitet.

## 1.3 Scrumteamet

Scrum definierar tre roller inom teamet. Scrum master, produktägare och utvecklare. Varje roll har sitt ansvarsområde. Scrum master håller metoden levande och ser till att arbetet är effektivt. Produktägaren ansvarar för backloggen och produktmålen. Utvecklarna utvecklar.

Scrumteamet är ansvarigt för alla produktrelaterade aktiviteter. Från samarbetet med intressenter, utveckling, tester, sammanställning av inkrement och färdig produkt.

Det finns ingen specifik arkitektroll. Ansvaret för intern arkitektur ligger på utvecklarna. Arkitekter som inte ingår i teamet är intressenter och samarbetar med teamet på samma sätt som andra intressenter.

### 1.3.1 Utvecklare

*Utvecklare är medlemmarna i Scrumteamet som engagerat medverkar till att skapa ett användbart inkrement i varje sprint.*

Bland utvecklarna finns inga roller utpekade men för arbete med vertikal arkitektur behöver ett antal kompetenser finnas förutom allmän utvecklarkompetens.

#### 1.3.1.1 Arkitektur

Arkitekturen i applikationen är ett delat ansvar inom utvecklargruppen. Alla utvecklare behöver vara väl förtrogna med implementationen av vertikal arkitektur och kopplingen till aktiviteter i backloggen.

Varje vertikal är sin egen implementation och därför är det viktigt att utvecklarna har förståelse för hur en bra struktur byggs inom vertikalen. Refaktorisering är en nyckelkompetens.

Förutom vertikalerna finns komponenter som hanterar delad teknik. Till exempel säkerhet, loggning och databashantering. Även om varje vertikal är sin egen berättelse är det viktigt att man följer gemensamma mönster för implementation och tester.

#### 1.3.1.2 Teknisk miljö

En agil utvecklingsmiljö är ofta DevOps-baserad. Det finns versionshantering av källkod och andra dokument. Nya versioner byggs automatiskt och i samband med förändringar körs automatiserade tester. Färdiga och automatiserade rutiner för installation (deploy) till test-, demo- och produktionsmiljö. Scrumteamet måste vara kunnig inom den miljön.

#### 1.3.1.3 Metodik

Använd automatiserad testning. Följ metodiken för Testdriven utveckling (TDD) där testerna skrivs före implementationen. Tester skall inte implementeras som ett separat steg efter implementationen.

Kodgranskning skall göras för att upprätthålla kvalité och likartad implementation mellan vertikaler.

#### 1.3.1.4 Kommunikation med intressenter

Utvecklarna har ett absolut ansvar när det gäller kommunikationen med intressenter och är måna om att hålla dialogen levande. Visa och förklara. Svara på frågor och var lyhörd för återkoppling. Var snabb att följa upp önskade förändringar. Engagerade intressenter är en nyckelfaktor för lyckad utveckling. Förstör inte relationen genom att vara otillgänglig, trög eller tråkig.

### 1.3.2 Produktägaren

*Produktägaren ansvarar för att maximera värdet av produkten som resultatet av utvecklingsteamets arbete.*

Produktägaren är en aktiv medlem i teamet. Hen har det slutgiltiga ansvaret för produktmålet och är väl införstådd med intressenternas vision och önskemål. Kommunikationen med intressenter och övriga teamet är levande och ständigt pågående. Produktägaren är drivande för att upprätthålla kontaktvägar mellan team och intressenter.

Backloggen är produktägarens ansvar. Backloggen skall vara en mappning mellan användarnas berättelser (user stories) och vertikaler. Under arbetets gång bryts posterna ner till lämpliga stycken för implementation. Backloggen byggs i nära samarbete med intressenter och team.

Produktägaren håller i sprintgranskningen och ansvarar för att den blir ett tillfälle där intressenter och team möts i engagerande diskussioner.

### 1.3.3 Scrum Master

*Scrum Mastern är ansvarig för etableringen av Scrum så som det är definierat i Scrumguiden. Detta görs genom att hjälpa samtliga att förstå både Scrums teori och det praktiska utövandet, inom såväl Scrumteamet som organisationen.*

Scrum master är ansvarig för implementation av Scrum och att anpassningarna av den lokala metodiken följer definitionen av Scrum.  Scrum mastern förstår innebörden i vertikal arkitektur och hur det passar in i Scrum-metodiken.

Med vertikal arkitektur är det en stark fokus på integrationen med intressenter. Scrum master har ett delat ansvar med övriga teamet för att hålla samarbetet levande.

## 1.4 Scrumaktiviteter

Scrum definierar en serie aktiviteter

### 1.4.1 Sprinten

*Hjärtat i Scrum är en Sprint, där idéer omsätts till värde.*

En sprint fokuserar på att leverera synliga värden. Det kan vara skisser, prototyper eller färdiga funktioner. Övergång från skiss till färdig funktionalitet görs genom poster i backloggen.

### 1.4.2 Sprintplaneringen

*Sprintplaneringen påbörjar Sprinten genom att fastställa arbetet som skall utföras i Sprinten. Den resulterande planen skapas av gemensamt av hela Scrumteamet.*

Under sprintplaneringen kopplas användarnas berättelser mot planer på implementation i vertikaler. Sprintplaneringen har fokus på att definiera mål för sprinten. Intressenternas berättelser mappas mot vertikaler och poster i backloggen väljs ut för implementation.

### 1.4.3 Dagligt Scrummöte

*Syftet med det dagliga Scrummötet är att granska framstegen mot Sprintmålet och att anpassa Sprintbackloggen vid behov, samt att justera arbetsplaneringen.*

Vertikal arkitektur påverkar inte det dagliga mötet på något avgörande sätt. Genom att utvecklingen görs inom vertikaler kan behovet av samordning mellan lager inte vara lika stort.

### 1.4.4 Sprintgranskning

*Syftet med en Sprintgranskning är att granska Sprintens utfall och att besluta om kommande anpassningar. Scrumteamet presenterar arbetsresultaten för intressenterna och framstegen mot Produktmålet diskuteras.*

Sprintgranskningen är hjärtat i kommunikationen med intressenter. Genom den vertikala arkitekturens fokus på komplett funktionalitet finns det som regel alltid något nytt att diskutera.

Undvik att göra mötet till en demo eller en uppräkning av vad som är nytt i senaste versionen. Visa nya funktioner som en del i användarberättelsen och var inte rädd att förklara kompletta sammanhang. Viss repetition kan vara bra för att skapa sammanhang för nyheterna. Ställ frågor till intressenterna och var noga med att notera synpunkter. Diskutera tankar och idéer. Ständig återkoppling och önskemål om förändringar är basen för agil utveckling.

### 1.4.5 Sprintretrospektiv

*Syftet med Sprintretrospektiven är att planera tillvägagångsätt för ökad kvalitet och effektivitet.*

Fundera på samordningen mellan vertikaler. Eftersom varje vertikal är sin egen berättelse är det lämpligt att fundera på samordningsvinster. Utvärdera behovet av delad funktionalitet och gemensamma rutiner. Hitta gemensamma lösningar på liknande problem. Uppdatera backloggen vid behov.

Diskutera hur interaktionen med intressenterna kan förbättras.

## 1.5 Scrumartefakter

### 1.5.1 Produktbacklogg

*Produktbackloggen är en framväxande, rangordnad lista, över vad som behövs för att förbättra produkten. Det är den enda källan till arbetsuppgifter Scrumteamet skall utföra.*

Backloggen skall vara tillgänglig och engagerande för intressenter. Prioritering av poster i backloggen är ett pågående arbete och är en viktig grund för planeringen av en sprint.

Vid arbetet med backloggen är det naturligt att utgå från användarnas berättelser och bryta ner i mindre delar som kan implementeras separat.  Posterna i backloggen skall motsvara användarnas krav. Vid nedbrytningen är det viktigt att inte förlora sammanhanget. Posterna i backloggen skall alltid kopplas till en vertikal.

En lämplig struktur på backloggen är att ha en hierarki där en nivå matchar implementationen av en vertikal och en lägre nivå där poster som implementeras är oberoende av andra poster. En komplett vertikal implementeras ofta inte i en enskild sprint utan den byggs en bit i taget

Ett bra sätt att strukturera arbetet är att inspireras av Ivar Jacobsons arbete med "Use Case". Ett use case kan implementeras som en vertikal. Ref [JACOB].

Backloggen innehåller framför allt önskemål från intressenter men det är lämpligt att avsätta en del av varje sprint till förbättringar och utveckling gemensamma rutiner som inte direkt är synliga för intressenterna. Interna förbättringar är viktiga men får inte ta fokus. Varje sprint skall skapa mervärde för intressenterna.

### 1.5.2 Åtagande: Produktmål

*Produktmålet beskriver ett framtida tillstånd för produkten som kan tjäna som ett mål för Scrumteamet att planera sitt arbete och sin verksamhet mot. Produktmålet finns i Produktbacklogen.*

Glöm inte att lyfta fram produktmålet i kommunikationen med intressenter. Det skapar ett sammanhang och en vision om målet för utvecklingen.

### 1.5.3 Inkrement

*Ett Inkrement är ett tydligt steg mot Produktmålet. Varje Inkrement är ett tillägg till samtliga tidigare Inkrement och grundligt verifierat, vilket säkerställer att alla Inkrement fungerar tillsammans. För att tillföra värde, måste Inkrementet vara användbart.*

I en vertikal miljö byggs inkrement på samma sätt som med en traditionell arkitektur. Skillnaden är att inkrement oftare skapar mervärde för intressenter. Behovet att bygga grundläggande strukturer minskar när vertikalerna själva implementerar sina behov.

### 1.5.4 Sprintbacklogg

*Sprintbackloggen består av Sprintmålet (varför), Produktbackloggens sprintposter (vad), såväl som en utförbar plan för att leverera Inkrementet (hur).*

Det är viktigt att backloggen speglar intressenternas behov. Ibland kan det vara nödvändigt för Scrumteamet att lägga in aktiviteter och då är det viktigt att intressenterna är involverade och förstår behovet.

### 1.5.5 Åtagande: Sprintmålet

### 1.5.6 Inkrementet

*Ett Inkrement är ett tydligt steg mot Produktmålet. Varje Inkrement är ett tillägg till samtliga tidigare Inkrement och grundligt verifierat, vilket säkerställer att alla Inkrement fungerar tillsammans. För att tillföra värde, måste Inkrementet vara användbart.*

Här briljerar den vertikala arkitekturen genom tydligt fokus på kopplingen mellan poster i backloggen och implementation.

### 1.5.7 Åtagande: Definitionen av Klar

*Definitionen av Klar är en formell beskrivning av tillståndet Inkrementet måste uppfylla med avseende på produktens kvalitetskrav.*

Exempel på ”definition of done”

- Komplett implementation av funktionen med tillhörande tester.
- Koden granskad.
- Tester klara. Automatisk och manuell testning genomförd.
- Status definierad (klar för release/prototyp/skiss etcetera).

Det är tillåtet att leverera prototyp-funktionalitet för att ge möjlighet till tidig feedback så länge det är tydligt definierat.