UML

**Ühtne modelleerimiskeel**, lühendina **UML**, ka **unifitseeritud modelleerimiskeel**, **UML-keel**, **ühtne mudelikeel**, ühtne visualiseerimiskeel (inglise keeles *Unified Modeling Language* ehk *UML*) on üldotstarbeline [noteeringukeel](https://et.wikipedia.org/w/index.php?title=Noteeringukeel&action=edit&redlink=1) keerulise [tarkvara](https://et.wikipedia.org/wiki/Tarkvara), peamiselt suurte objektorienteeritud projektide spetsifitseerimiseks ja visualiseerimiseks. UML põhineb sellistel vanematel noteeringumeetoditel nagu [Booch](https://et.wikipedia.org/w/index.php?title=Booch&action=edit&redlink=1), [OMT](https://et.wikipedia.org/w/index.php?title=OMT&action=edit&redlink=1) ja [OOSE](https://et.wikipedia.org/w/index.php?title=OOSE&action=edit&redlink=1).

### **UML struktuuriskeemid**

Struktuuriskeemid struktuurimudelite koostamiseks:

* Klassiskeem (*class diagram*)
* Komponendiskeem (*component diagram*)
* Liitstruktuuriskeem (*composite structure diagram*)
* Evitusskeem (*deployment diagram*)
* Objektiskeem (*object diagram*)
* Paketiskeem (*package diagram*)

#### **Klassiskeemid**

Klassiskeemid (*class diagram*) moodustavad keskse osa OO meetodite puhul, sest klasside kasutamine on OO meetodite aluseks. Klassiskeem kuulub staatiliste skeemide hulka. Klassiskeem kirjeldab objektide tüüpe süsteemis ja staatilisi seoseid nende vahel. Samuti näitab klassi omadusi e atribuute, klassi operatsioone ja piiranguid objektide vahelistele seostele. Klassi omaduste ja operatsioonide ühisnimetajaks on UML-is „feature“, mida on tõlgitud kui erisus.

Klassiskeemi kõige olulisemad osad on klassid ja nende vahelised seosed, siis võib öelda nii, et klassid ja objektid kirjeldavad süsteemi elemente ja seosed kommunikatsiooni ehk suhtlemist. Klasside kasutamine keeruliste süsteemide kirjeldamisel on tegelikult vana nipp, mida inimkond juba tuhandeid aastaid kasutanud on.

#### **Klassid ja objektid**

Objektid eksisteerivad reaalses maailmas (või arusaamises maailmast). Objekt saab olla osa ükskõik millisest süsteemist. Objektid tarkvarasüsteemis muutuvad teoreetiliseks, neid saab moodustada tegelike objektide omadusi ja käitumist analüüsides. Objektides peegeldub inimese arusaamine maailmast. Klass kirjeldab objekti tüübi omadusi ja käitumist. Tarkvarasüsteem loob klassist objektide eksemplarid ehk isendid. Objekti ja klassi vahel on sarnane seos nagu muutuja ja andmetüübi vahel.

#### **Süsteemi mudelid ja vaated**

Klassiskeeme joonistatakse kogu analüüsi ja kavandamise jooksul ja neid joonistatakse erinevatel tasemetel. Erinevad mudelid, mida klassiskeemi abil joonistatakse on: Kontseptuaalne e. mõistemudel – skeemil kasutatakse uuritava valdkonna mõisteid. Sellel skeemi näidatakse seoseid, mis kehtivad elus. Ehkki need mõisted võivad vastata ka hilisemale realisatsioonile, ei pruugi nad alati üks-üheses vastavuses olla. Mõistevaade tuleb joonistada ilma mõtlemata samal ajal realisatsioonile ja kindlasti programmeerimiskeelest sõltumatult. Mudel on vajalik valdkonnast arusaamiseks. Mõistemudel peaks samal ajal ka kliendile arusaadav olema. See ei pea olema süsteemi täpne mudel, rohkem see, kuidas tegelikust süsteemist aru saadakse. Olenevalt arendusmetoodikast sobib seda nimetada ka valdkonnamudeliks (*domain model*). Spetsifikatsioonimudel – sellel mudelil peetakse silmas tarkvara liidest, vaadatakse tüüpe, mitte objekte. Aluseks on OO poolt tehtav range eristus objekti liidesele (see, mille kaudu saab objekti kasutada, mida näevad teised objektid) ja realisatsioonile. Klassi tüüp on tegelikult kui liides, millel võivad olla erinevad realisatsioonid sõltuvalt keskkonnast, vajalikust jõudlusest, programmeerijast, jne. OO-keeled seovad liidese ja realisatsiooni ja seepärast vahe nende vahel jääbki teinekord märkamata. Realisatsioonimudel – põhineb realisatsioonile ja seda vaadet kasutatakse kõige enam. Siin on juba tõesti klassid ja keskendutakse sellele, kuidas klassid realiseerida. Realisatsioonimudelist võib olla lootust ka mingit koodi genereerida (eesmärk – teha seda võimalikult põhjalikult). Klasside skeleti genereerimisega saavad juba tänased CASE-vahendid täiesti edukalt hakkama. Kui klassiskeemi joonistatakse, peab kogu skeemi ulatuses kasutusel olema sama vaatenurk, st mudelil peavad kõik klassid olema samal tasemel. Valdkonnamudelile ei tohi sattuda realisatsiooni detailid. Vastasel juhul ei ole mudelit võimalik õigesti tõlgendada. Vaated ei kuulu UMLi alla, sest tegemist on juba kasutusmeetodiga ja mitte skeemi endaga. Skeemi joonistades on oluline vahet teha, millisel vaatel ta põhineb. Ühel arendusetapil joonistatakse üks vaade ja teisel etapil teine vaade. Milleks klassiskeemid? Klassiskeemid on alati kasulikud, kui on tegemist OO-ga. Hea, kui saab piirduda kirjeldatud võimalustega ja ei pea kasutama nn edasijõudnute arsenali. Kes sellest teada tahab, uurigu ise kirjandusest edasi. Igal juhul ei ole vaja kogu klassiskeemi rikkust ühele konkreetsele skeemile paigutada. Olulisemad osad on seega: klassid, sidemed, atribuudid, üldistus, piirangud. Skeem tuleb joonistada konkreetsest vaatest lähtudes: Olles analüüsietapil, joonistatakse mõistevaade ja valdkonnavaade. Sellest tuleb igasugune realisatsioon eemal hoida. Olles projekteerimise juurde jõudnud, joonistatakse spetsifikatsiooni vaade. Realisatsioonivaade jäetakse vahel hoopis tegemata või tehakse ainult mingite osade jaoks. Parem vähem õigeid skeeme, kui hunnik vananenud skeeme kogu süsteemi kohta.

#### **Komponendiskeem**

UML1 tegi suuremat vahet skeemil. Vastavalt UML2-le näeb komponendiskeem välja sarnane klassiskeemile. Eristamaks klassidest, saab kasutada väikest komponendiikooni klassi kasti sees või kasutada stereotüüpi <<component>>. Skeemil pole midagi uut. Komponente ühendatakse liideste kaudu, kasutatakse kuuli ja pesaga tähistust. Komponendid kui tarkvarasüsteemi osad peavad olema sõltumatult ostetavad ja värskendatavad seega jaotamine komponentideks on sama palju turu otsus (mida tasub osta/müüa) kui tehniline otsus.

#### **Liitstruktuuriskeem**

Uus skeem UML2-s. Võimalus jaotada klass hierarhiliselt oma seesmise struktuuri järgi. Võib võtta keerulise objekti ja jagada ta osadeks. Näidatakse klassi seesmist struktuuri ja millised osad vajavad ja pakuvad milliseid liideseid. Selleks kasutatakse juba tuttavat pall ja pesa-esitust. Klassi kasti sisse joonistatakse järgmised kastid, mis tähistavad osi. Klassi liidesed (pakutavad ja vajatavad) saab ühendada vastavate osade külge. Osade puhul kasutatakse ka mitmesuse näitamise võimalust. Asjatundjad arvavad, et need on väga vajalikud, kuid esialgu esitatavad kirjeldused on äärmiselt lakoonilised.

#### **Evitusskeem**

Evitusskeem näitab füüsilisi seoseid tarkvara ja riistvara komponentide vahel. Sellega saab näidata, kuidas kogu tarkvara mööda hajussüsteemi laiali paisatud on. Näitab kogu süsteemi topoloogiat – millised on seadmed, töötavad keskkonnad ja tehised, mis on selles arhitektuuris. Saab vaadata igasse sõlme ja näha, mis selles sõlmes paikneb.

##### **Komponendid**

Sõlm (node) – tükike riistvara, alates lisaseadmest ja lõpetades suurarvutiga. Sellel saab paikneda tehis. Sõlmel saavad olla alamsõlmed (subnodes), mis võivad olla töötavad keskkonnad. Sõlm võib olla nii tüüp (nt teatud tüüpi protsessoriga arvuti) kui ka isend (üks konkreetne arvuti). Konkreetsete omaduste kirjeldamiseks on sõlmel atribuudid. Teinekord võib osutuda vajalikuks näiteks arvuti konfiguratsiooni kirjeldamine – sellel skeemil saab ka seda teha. Sõlme iseloomustamiseks võib kasutada stereotüüpe <<device>> ja <<execution environment>>. Seega sõlm võib olla ühelt poolt konkreetne seade ja teiselt poolt käituskeskkond (operatsioonisüsteem, andmebaasisüsteem jms) Sõlme juures võib näidata tema kohta igasuguseid omadusi: tootja, opsüsteem, vastavat tüüpi sõlmede arv. Viimase jaoks ei ole eritähistust ette nähtud. Seosed sõlmede vahel (*connection*) – millised on suhtlusteed, mida mööda süsteemis info liigub. See näitab, et sõlmed suhtlevad üksteisega, saates üksteisele objekte või sõnumeid. Võib lisada stereotüübi, mis näitab, millise protokolli abil suheldakse, nt <<TCP/IP>>. Tehised (*artifacts*) – Tehised saavad olla mööda sõlmi laiali paisatud. Nad kirjutatakse sõlmede sisse. Pikemate kirjelduste jaoks võib kasutada eraldiseisvat <<deployment spec>>-i. St pikemat kirjeldust ei ole mõistlik skeemile lisada ja spetsifikatsiooni võib eraldi lugeda.

#### **Objektiskeem**

Objektskeem on teatud tüüpi klassiskeem. Objekt esitab klassi olekut teatud ajahetkel süsteemi töö käigus. Objektiskeem esitab süsteemi erinevate klasside olekuid ning nendevahelisis relatsioone või assotsiatsioone teatud ajahetkel.

#### **Paketiskeem**

Pakett on konstruktsioon, mille abil saab võtta suvalise UML-i konstruktsiooni ja grupeerida tema elemendid üldisemal tasemel. Paketiskeemiga näidatakse tavaliselt klasside kogumeid (pakette) ja sõltuvusi nende vahel. Kuid paketti on võimalik kasutada ka teiste UML-i osade juures. Idee poolest saab iga klass UML-mudelis kuuluda vaid ühte paketti, kuid pakett võib kuuluda järgmisesse paketti. Tekib hierarhiline struktuur ja ühes paketis võivad olla koos teised paketid ja üksikud klassid. Programmeerimiskeeltes tagatakse võimalus programmi jagada füüsiliselt erinevatesse failidesse. Sõltuvalt keelest kutsutakse seda konstruktsiooni erinevalt. Mõnel puhul ka paketiks. Iga pakett on eraldi nimeruum (*namespace*), st klassidel ühes paketis peavad olema unikaalsed nimed. Sama nimega erinevad klassid võivad aga kuuluda erinevatesse pakettidesse. Iga nimi peab kuuluma mingisse nimeruumi. Kui on vaja üheselt näidata, kuhu paketti nimi kuulub, kasutatakse pikka nime (fully qualified name), iga osa vahel on koolonid.

#### **Skeemi komponendid**

Paketid (*package*) – ristkülikud, kus on kirjas ainult paketi nimi või ka temas sisalduvad klassid ja/või teised paketid. Viimased võivad olla välja joonistatud või ainult nimekirjana. Klassid võivad paketis olla *public* või *private*. Paketi liides tuleb teha võimalikult väike ja eksportida vaid nende avalike klasside need meetodid, mida vaja on. Kuidas valida, millised klassid ühte paketti kokku panna? Üheks soovituseks on järgmised kaks printsiipi:

* Ühise sulundi printsiip (*Common Closure Principe*): pane kokku klassid, mida tuleb muuta samadel põhjustel.
* Ühise taaskasutuse printsiip (*Common Reuse Principe*): ühes paketis olevaid klasse tuleks koos korduvkasutada.

Kuid mitmed põhjused on seotud ka klasside vaheliste sõltuvustega.

Sõltuvus (*dependency*) on seos kahe mudelielemendi vahel, kus muudatus ühes (sõltumatus) mudelielemendis mõjutab teist (temast sõltuvat) mudelielementi. Sõltuvust kujutatakse katkendnoolena kahe paketi vahel. Sõltuvus võib eksisteerida ka kahe klassi vahel, kui ühe klassi kirjelduse muutmine võib põhjustada muudatusi teises klassis. Sõltuvust saab põhjustada see, kui:

* üks klass saadab teisele klassile teate;
* üks klass kuulub teise klassi andmete hulka;
* ühe klassi jaoks on teine klass mõne meetodi parameetriks.

Sõltuvus on alati suunatud ja ei ole transitiivne, st kui klass1 sõltub klass2st, siis automaatselt klass2 ei sõltu klass1st. Reaalselt võib ka vastassuunas sõltuvus tekkida, kuid siis on tegemist tõsiste disaini probleemidega. Heaks tavaks on süsteemi ülesehitamisel tema kasutajaliidese ja sisu eraldamine. Tavaliselt on sõltuvus sel juhul suunatud kasutajaliideselt sisu poole, st kui muutub töötava süsteemi sisu, tuleb muuta ka kasutajaliidest, kuid kasutajaliidese muutmine ei tohiks põhjustada süsteemi muutmist.

Pakettide vahel on samuti sõltuvused. Esituspaketi ja valdkonnapaketi vahel on sõltuvus siis, kui ükskõik millisel klassil esituspaketis on sõltuvus ükskõik millisest klassist valdkonna paketis.

Kasutatud allikad:

<https://en.wikipedia.org/wiki/Unified_Modeling_Language>

https://et.wikipedia.org/wiki/%C3%9Chtne\_modelleerimiskeel