

**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Gantt diagram alkalmazás fejlesztése HTML 5 platformon

Készítette

Balog Ábel PéterKonzulens

Albert István

2016

Tartalomjegyzék

[Összefoglaló 5](#_Toc449979630)

[Abstract 6](#_Toc449979631)

[1. Bevezetés 7](#_Toc449979632)

[1.1. Gantt diagram 7](#_Toc449979633)

[1.2. Létező megoldások 8](#_Toc449979634)

[2. Vékonykliens fejlesztés 9](#_Toc449979635)

[2.1. Bevezetés 9](#_Toc449979636)

[2.2. JavaScript 9](#_Toc449979637)

[2.3. TypeScript 10](#_Toc449979638)

[2.4. HTML5, Canvas, KonvaJS 12](#_Toc449979639)

[2.5. AngularJS 12](#_Toc449979640)

[2.5.1. Angular Material 14](#_Toc449979641)

[3. Gantt diagram 16](#_Toc449979642)

[3.1. Bevezetés, követelmények 16](#_Toc449979643)

[3.2. Modell 16](#_Toc449979644)

[3.2.1. Projekt felépítése 17](#_Toc449979645)

[3.2.2. Munkanaptár 18](#_Toc449979646)

[3.2.3. Ütemező 18](#_Toc449979647)

[3.2.4. Erőforrások 19](#_Toc449979648)

[3.3. Ütemezés, CPM 19](#_Toc449979649)

[3.3.1. Bevezetés 19](#_Toc449979650)

[3.3.2. Kritikus útvonal módszer 19](#_Toc449979651)

[3.3.3. Erőforrások ütemezése 20](#_Toc449979652)

[3.4. Megvalósítás 22](#_Toc449979653)

[4. A teljes webalkalmazás 25](#_Toc449979654)

[4.1. Bevezetés, követelmények 25](#_Toc449979655)

[4.2. Architektúra 25](#_Toc449979656)

[4.3. Megvalósítás 26](#_Toc449979657)

[5. Összefoglalás 27](#_Toc449979658)

[Ábrák jegyzéke 28](#_Toc449979659)

[Irodalomjegyzék 30](#_Toc449979660)

Hallgatói nyilatkozat

Alulírott Balog Ábel Péter, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2016. 04. 30.

Balog Ábel Péter

# Összefoglaló

A feladatom egy olyan webalkalmazás elkészítése, ami segíti a projekt menedzsereket abban, hogy egy projekt során az egyes részfeladatokat az erőforrásokhoz hozzárendelje és ezeket egy Gantt diagram segítségével megjelenjen. Erre a feladatra vastagkliens megoldások (pl: Microsoft Project) már léteznek, de a webes technológiák fejlődésével már van lehetőség hasonló funkcionálatisú, böngészőben futó alkalmazás elkészítéséhez.

Az elkészítendő alkalmazás fő felülete egy Gantt diagram lesz, ahol a projektvezető kap egy áttekintést arról, hogy az egyes feladatok hogyan kövesség egymást

# Abstract

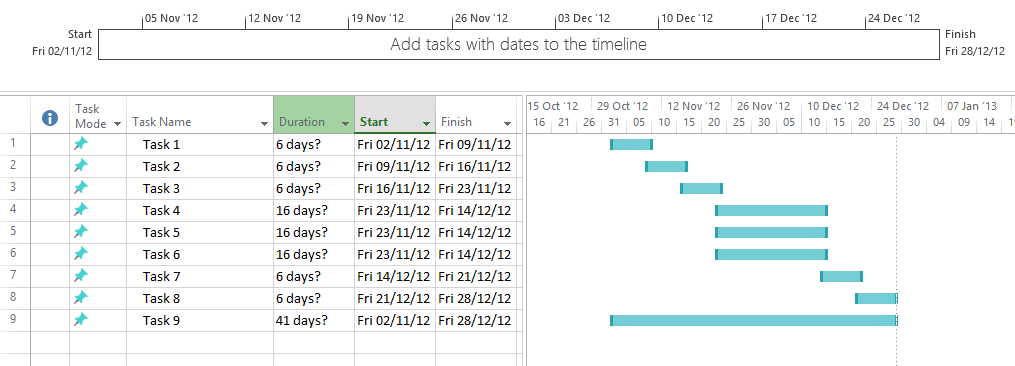
English abstract of the thesis work. This summarises the content of the thesis in 0.5–1 pages and is uploaded to the Thesis Work Portal as well.

# Bevezetés

## Gantt diagram

A Gantt diagram a projekt menedzsment egyik alapvető eszköze, ami vizuálisan, könnyen érthető módon és átláthatóan megjeleníti az egyes feladatok a projekt során, azok időszükségleteit, illetve hogy mikorra is ütemezzük annak elvégzését.

Megjelenésre egy oszlop diagramhoz hasonlítható, aminek egyes oszlopai jelölik, hogy az adott feladatot mikor kell elvégezni az ütemezés szerint.



1. ábra Gantt diagram a Microsoft Project 2013 programban

Karol Adamiecki volt az első, aki ilyen diagramot készített 1896-ban, de eredményeit nem publikálta. Később Henry Gantt volt az, aki tőle függetlenül kitalálta a ma ismert Gantt diagramot, amit róla neveztek el. Később Adamiecki is publikálta eredményeit, de azok csak Lengyelországban váltak ismertté.

Az informatika előretörésével kapott újra nagy hangsúlyt a diagram, hiszen a számítógépek rendkívül gyorsan tudták a feladatokat ütemezni, és azokat vizuálisan megjeleníteni.

Ma egy projektmenedzser munkájának jelentős része a projekt feladatainak ütemezésével, erőforrások feladathoz rendelésével és a feladatok közötti függőségek felismerésével telik, amikhez a Gantt diagram és a köré épült megoldások adják az eszközöket.

## Létező megoldások

Az egyik legelterjedtebb létező megoldás a Microsoft Project, így a dolgozat folyamán ezt tekintem mintának. Az 1. ábrán látható, hogy hogyan épül fel a felülete. Az egyes feladatok egymás alatt jelennek meg további információk társaságában (időtartam, kezdés, befejezés időpontja). Bal oldalt látható vizuálisan, hogy mely feladatokat mikor kell elvégezni. Látható, hogy az egyes feladatok csoportba rendezhetőek, és köztük függőségeket tudunk definiálni, amik vizuálisan is megjelennek.

Fontos szempont, hogy a diagram jól átlátható, az egyes feladatok jól elkülöníthetőek egymástól, és egy külső szemlélő számára is könnyen átlátható, hogy az adott projekt milyen részfeladatokból áll, azok között milyen tartalmazási és függőségi relációk vannak, illetve mikor kell őket elvégezni.

A diagramnak nem célja, hogy leírja az egyes feladatok részleteit, technológiai követelményeit, ezek az információk mind az átláthatóság kárára tudnának csak megjelenni, emiatt ezekkel kapcsolatban semmilyen információt nem hordoz.

# Vékonykliens fejlesztés

## Bevezetés

Az alkalmazás vékonykliens környezetben kell, hogy fusson, HTML5 és JavaScript technológiák segítségével. Hátránya, hogy a legtöbb esetben lassabb, mint a natív megoldások, viszont cserébe platform független, azaz minden környezetben futni fog, mindössze egy böngészőre van szükség. Mivel a feladat kiírásában szerepel, hogy ez egy önállóan futni képes alkalmazás legyen, ezért az üzleti logika teljes része kliens oldalon kell, hogy fusson.

## JavaScript

A fejlesztéshez a JavaScript nyelvet választottam. A támogatottsága nagyon széles a böngészők körében, illetve futtatásához nem szükséges külön beépülő modul.

A JavaScript-et 1997-ben ECMAScript néven szabványosították, itt írták le először egységesen a nyelv alapjait. Alapvetően egy objektum alapú nyelvről van szó, azaz tartalmaz objektumokat, de procedurálisan is lehet programozni. A JavaScript objektum felfogása erősen eltér a szokásos nyelvekétől (C++, Java, C#), úgynevezett prototípus alapú nyelvről beszélünk. Minden objektum egy prototípus példánya, de azt futási időben tetszőleges módosíthatjuk, adhatunk hozzá metódusokat, tagváltozókat, módosíthatjuk értéküket, típusukat, implementációjukat.

A nyelvben van beépített garbage collector, így nem kell különösen nagy figyelmet fordítani a memóriaszívárgásra, mindezek ellenére segíthetjük a futtató környezetet, például egy váltózó kinullozásával jelezhetjük, hogy már nem tartunk igényt az értékére.

## TypeScript

Igaz, hogy a JavaScript nyelv prototípus alapú objektum megközelítése nagyon jól támogatja a bővíthetőséget, de fejlesztési időben az egyes változók típusaival kapcsolatban semmilyen megkötés sincs, így a függvények, objektumok közötti kommunikációban, fordítási időben nem tudjuk megkövetelni a hívó féltől, hogy egy adott objektum rendelkezzen bizonyos tulajdonságokkal.

Ennek orvoslására több megoldás is született, amik különböző módokon segítik a fejlesztést. Az egyik ilyen a Microsoft által fejlesztett és karbantartott TypeScript. A fejlesztésnél fontos szempont volt, hogy lehetőleg hasonló szintaktikája legyen, mint a készülő EcmaScript 6-nak, ezzel segítve az esetleges későbbi átállást. A TypeScript nem egy új nyelv, csak egy plusz réteg a JavaScript felé, ami fejlesztési időben garantálja a típusosságot, JavaScript-re fordul, és futási időben már semmilyen szerepet sem játszik.

Egy egyszerű osztály így néz ki:

**module** Test {  
 **export class** Person {  
 **public name**: **string**;  
 **private parents**: Person[];  
   
 **constructor**() {  
 **this**.**name** = **""**;  
 **this**.**parents** = [];  
 }  
   
 addParent(person: Person): **void** {  
 **this**.**parents**.push(person);  
 }  
 }  
}

A *module* kulcsszóval tudjuk az osztályokat csoportokba rendezni, hasonló, mint a *Java*-ban a *package* vagy *C#*-ban a *namespace*. A modulon belül van lehetőség osztály deklarációra. Az *export* kulcsszó jelenti, hogy az osztály a modulon kívülről is elérhető. Az utána lévő kód nagyon hasonlít a szokásos programozási nyelvekben megismert szintaktikához, egyedül a típusdeklaráció más, kivételesen nem a változó neve elé írjuk a típust, hanem a változó után, kettősponttal elválasztva.

A fordító a TypeScript fájlokat lefordítja JavaScript nyelvre (esetleg létrehoz hozzá egy map fájlt, ami a debugger-nek segít, hogy melyik JavaScript kódrészlet melyik TypeScript kódrészlethez tartozik). Az így keletkezett fájlt már be tudjuk tenni a HTML-be, és a böngésző is képes a futtatására.

A TypeScript legnagyobb előnye, hogy kompatibilis a már létező JavaScript-ben írt könyvtárakkal. Ha valaminek az any típust adjuk, akkor az bármi lehet (azaz a fordító nem ellenőrzi, hogy a rajta meghívott metódusok, tagváltozók valóban részései az objektumnak), vagy pedig elég megadni a függvénykönyvtár interfészét, ami fejlesztési időben segít a kódkiegészítéssel, futási időben pedig már az interfész egyes kulcsszavai pontosan egy-egy JavaScript függvényt/típust takar. A nagyobb függvénykönyvtárak vagy saját maguk készítenek ilyen interfészleírást, vagy pedig a fejlesztői közösség készíti el őket, és teszik elérhetővé bárki számára.

A nagy előnye egy ilyen plusz réteg bevezetésének, hogy fejlesztés közben sokkal kevesebbszer kell belenézni a dokumentációba vagy a hívott kód implementációjába, hiszen a fejlesztési környezet az automatikus kiegészítéssel segíti a fejlesztést. Ha az egész alkalmazás TypeScript-ben íródott, akkor biztosak lehetünk abban, hogy ahol egy adott típust megkövetelünk, ott az is van, de ha már külső JavaScript kódból hívjuk, akkor szükséges az ellenőrzés.

Vannak esetek, amikor nem szeretnénk egy adott kódrészlethez osztályokat készíteni, hanem egy adott feladatot minél gyorsabban és egyszerűbben akarjuk megoldani kihasználva a Javascript prototípus alapú gondolkozásmódjának lehetőségét. Erre a Typescript ad lehetőséget, méghozzá az any típus bevezetésével. A fordító nem fogja ellenőrizni az ilyen típusú változóknál, hogy milyen metódusokat és attribútumokat ismer hozzá, hanem mindent továbbenged, mint pl a következő kódrészletnél:

var foo: any = {};

any.a = 2;

any.b = 3;

any.add((): any => {

return + b;

}

console.log(any.add()); //outputs 5

console.log(any.c) //no exception, outputs undefined

## HTML5, Canvas, KonvaJS

## AngularJS

A webalkalmazás működését a Google által fejlesztett AngularJS fogja segíteni. Kezdetekben MVC (Model-View-Controller) mintát követett, idővel viszont már sok MVVM-es (Model-View-Viewmodel) minta is része lett a technológiának, így most már az általuk kitalált MVW (Model-View-Whatever) mintát követik.

Ez a keretrendszer a fejlesztőt igyekszik abban segíteni, hogy a kezdektől fogva támogassa

Az AngularJS legnagyobb előnye, hogy a HTML bizonyos elemeit hozzá lehet kötni (binding – adatkötés) a mögötte álló Controller bizonyos elemeihez, és amíg az AngularJS kereteiben belül dolgozunk, addig a változások automatikusan nyomon lesznek követve.



Azért választottam ezt a keretrendszert az alkalmazáshoz, mert jól illeszkedik a HTML által kijelölt deklaratív programozási modellel, ezt pedig a direktívákkal (directives) bevezetésével éri el. Ezek a programnak olyan komponensei amik egy adott HTML elem működését módosítják, vagy új elemeket definiálnak. Például, ha a program alatt mi több helyen is használunk egy bizonyos HTML elemekből álló, egyedi viselkedéssel bíró komponenst, akkor ha létrehozunk neki egy direktívát utána elég megadni

Az AngularJS funkcionalitása elég sokrétű, ezért nem is témája a jelen dolgozatnak, de egy jól bővíthető, dinamikusan fejlődő függvénykönyvtárról van szó, ami segít abban, hogy az alkalmazás minél nagyobb része (jelen esetben az egész) JavaScript-ben kerüljön implementálásra. Tartalmaz template könyvtárat, REST klienst és rengeteg egyéb funkciót.

### Angular Material

Az Angular Material szintén a Google által fejlesztett függvénykönyvtár, ami az egységes, reszponzív felület kialakításában nyújt segítséget a fejlesztő számára.

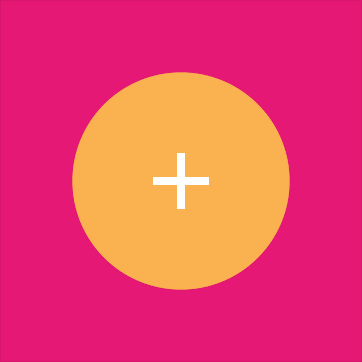
Az Android mobil operációs rendszerének Lollipop (5.0) kiadásában a Google egy új designelvet vezetett be, ezt hívják Material Design-nak. A következő alapelveket fekteti le:

* Az anyag mindennek az alapja: A designelemek alapvetően papírból (rétegek) épülnek fel, amiknek vannak színei. Ezek a rétegek egymáson helyezkedhetnek el, egymás fölött és alatt, van szélességük, magasságuk, de mélységük nincs, viszont a távolságuk a felhasználótól különbözhetnek egymástól (árnyékként jelenik meg). Fontos, hogy a felület megjelenése közel álljon a valósághoz, azaz például az egyes réteget egymást nem metszhetik.



2. ábra Material design – anyag

* Merészség, grafikusság, tudatosság: A felület színvilága legyen merész, megfontolt. Legyenek grafikus elemek, változó formák. Fontos, hogy a felületet tudatosan építsük fel, a felhasználónak a tekintetét a fontos helyekre „csábítsa”. A lényeg legyen kiemelve, ne kelljen keresgélni, a felület legyen szemcsalogató.



3. ábra Material Design – merészség

* Az animáció, mozgás fontossága: A felhasználók reakciója egy mozgásra általában egy inflekciós pontja, azaz a felhasználói műveletek a mozgások irányát módosítják. A mozgások folyamatosak legyenek, semmi se törje meg a folytonosságot, az animáció kezdeti szakaszát gyorsulás, a közepét egységes sebesség a végét pedig lassulás jellemezze.

A fenti elvek betartásában nyújt segítséget az Angular Material. Különböző modulokat tartalmaz, amik kész komponensként beépíthetőek a webalkalmazásba. A modulok beszúrása az AngularJS által kitalált direktívák segítségével működik, azaz vagy teljesen saját fejlesztésű HTML tag-eket használatával, vagy egyedi attribútumok használatával lehet testre szabni a megjelenést, viselkedést.

Segítséget nyújt, hogy az alkalmazás reszponzív működésű legyen, azaz a felület a képernyő vagy felbontás függvényében változzon, így igazodva például a mobil kijelzőkhöz.

# Gantt diagram

## Bevezetés, követelmények

Ebben a részben azt fogom kifejteni, hogy a diagramnak és ütemezésnek milyen követelményeknek kell hogy megfeleljen, és ezeket hogy valósítottam meg.

Mint, ahogy az 1. fejezetben kifejtettem a cél a Microsoft Project 2013 alapján egy Gantt diagram megjelenítő, kezelő és ütemező alkalmazás elkészítése. Első körben az ütemezés és a megjelenítés megvalósítását fogom ismertetni.

## Modell

C:\Users\Abel\Downloads\overview.png

A modell négy darab komponensből épül fel, amik egymást teljes mértékben ismerik, és szorosan kapcsoltak.

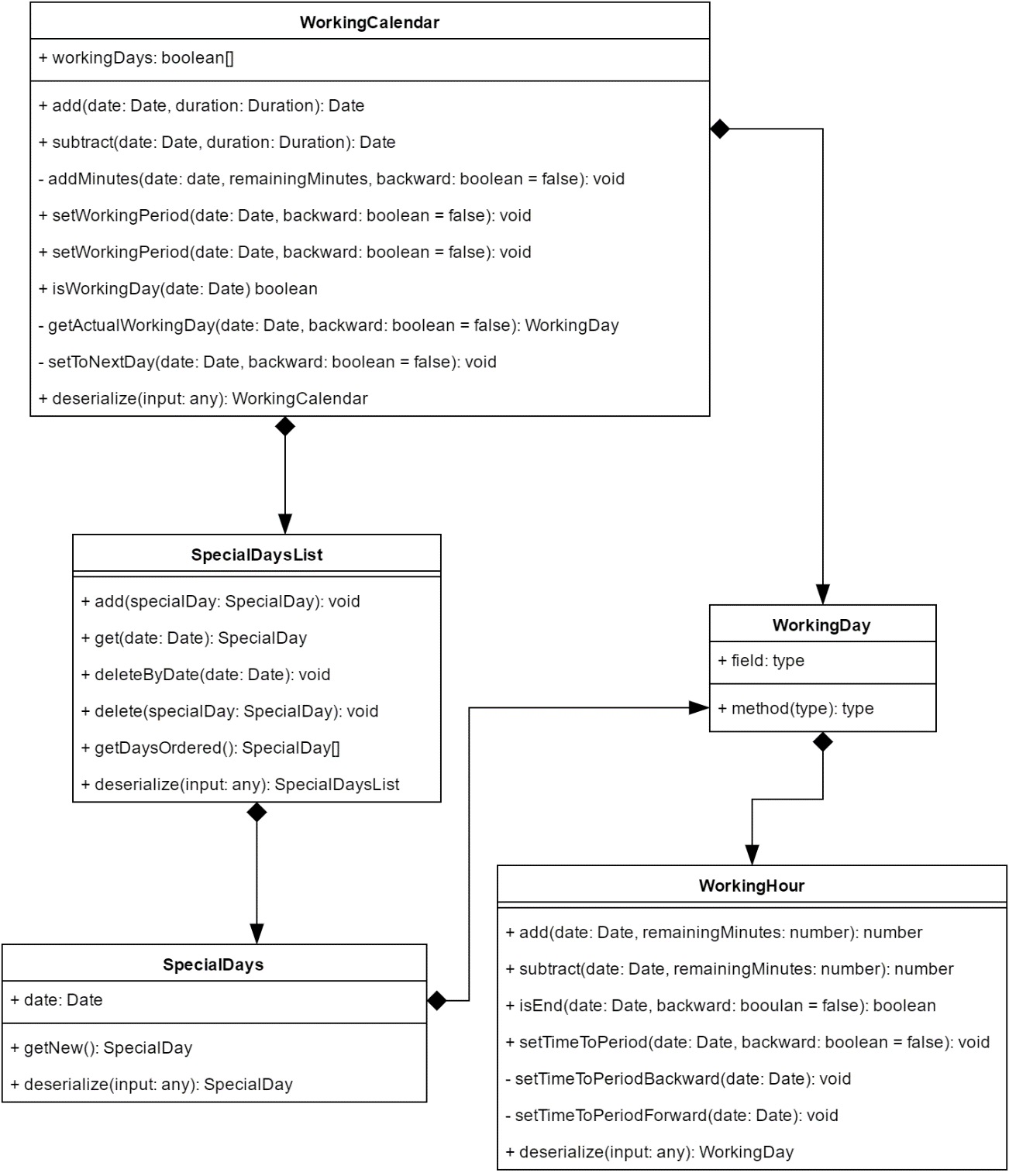
### Projekt felépítése

C:\Users\Abel\Downloads\model (2).png

A projekt és az abban lévő feladatok kapcsolata a fenti ábrán látható.

* Task: Egy, a projekt során megoldandó feladatot leíró absztrakt osztály (mivel TypeScript-ben nem lehet absztrakt osztályokat létrehozni, így elvileg ez egy normális osztály, de soha nem kerül példányosításra).
* Summary: Egy feladatcsoportot leíró osztály, ütemezés során nem használjuk a gyerekmet

### Munkanaptár



A fenti ábrán látható a munka- és szünnapokat kezelő naptárkomponens felépítése.

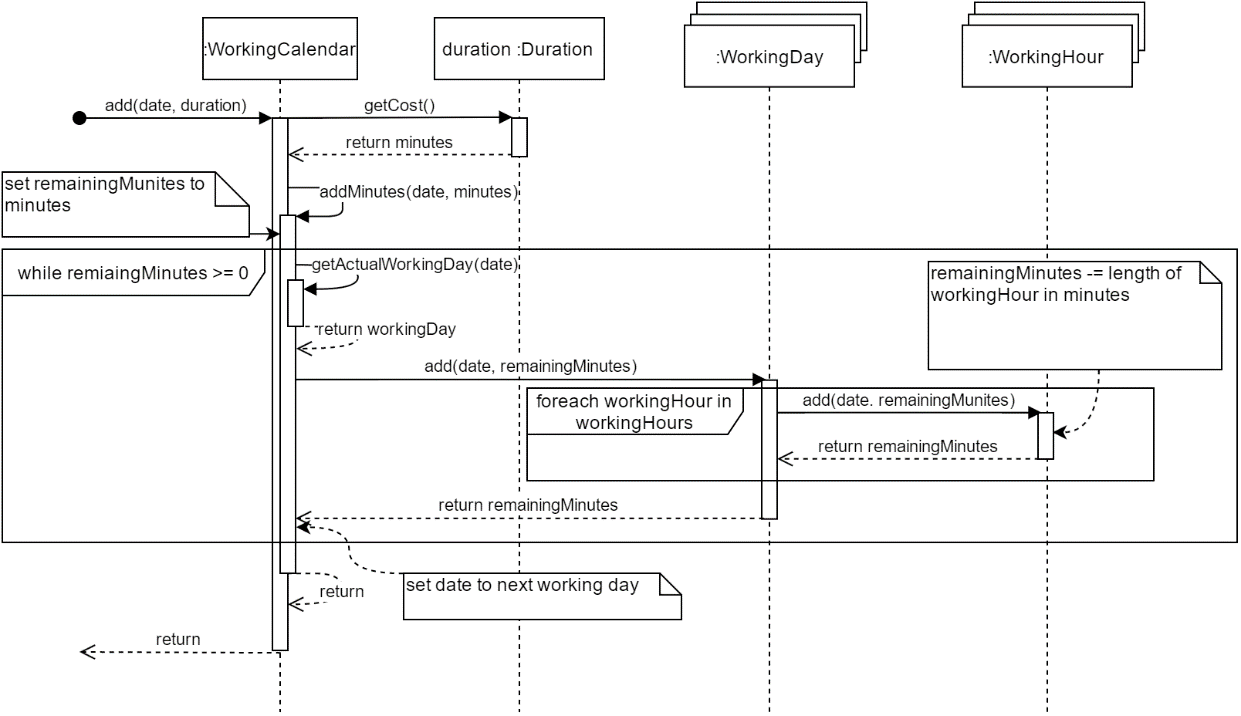
Az egyes osztályok feladata a következő:

* WorkingCalendar: Rajta keresztül érhető el a legtöbb művelet, ennek a komponensek ez a fő belépési pontja. Egy naptárt tart nyilván, minden naphoz lekérdezhető, hogy akkor éppen mi a munkarend, és ez alapján számításokat is lehet végezni, a pontos funkciókat majd a teljes komponens funkcionalitásakor fejtem ki.
* WorkingDay: Egy adott nap munkarendjét írja le, tartalmazza, hogy aznap mely periódusokban (WorkingHour) van munka (pl: 8-12, 12:30-16:30).
* WorkingHour: A munkanap során egy adott periódust leíró objektum, egy kezdő és egy végző órát tartalmaz.
* SpecialDay: Egy, a szokásos munkarendtől eltérő napot leíró osztály. Tartalmazza, hogy pontosan melyik napról van szó (date: Date attribútum), és hogy aznap milyen munkarend szerint van munka.
* SpecialDaysList: Egy listában tartalmazza, hogy milyen olyan napok vannak, amik eltérnek a szokásos munkarendtől, dátum alapján el lehet kérni tőle, hogy aznap speciális, és ha igen, akkor milyen munkaórák vannak (SpecialDay).
* Duration: Egy időtartamot leíró osztály, főleg az egyes feladatok hosszánál kerül felhasználásra, legkisebb általa kezelt érték a perc.

Az alaposztály a WorkingCalendar, amin keresztül a következő műveletek érhetőek el:

* Hozzáadás, kivonás: Egy adott dátumhoz tud hozzáadni vagy kivonni egy időtartamot (Duration), figyelembe véve a munkarendet. Gyakorlatilag ezek azok a műveletek, amik kiszámolják, hogy a feladat kezdésének időpontjának ismeretében a munkanaptár szerint mikor is van a befejezése. Ahhoz, hogy ezt a műveletet elérjük a WorkingCalendar osztály add és subtract metódusát kell meghívni.
* Speciális napok kezelése: A WorkingCalendar specialDaysList attribútumán keresztül tudjuk módosítani a speciális napok listáját, erre szolgálnak a SpecialDaysList osztály publikus metódusai.

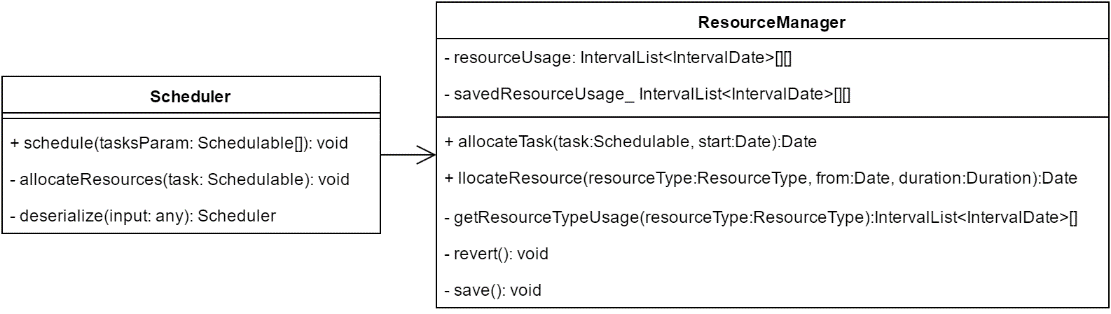
A legbonyolultabb művelet a hozzáadás és kivonás, ezért ennek működését külön, részletesebben is kifejtem:



A fenti szekvenciadragrammon összefoglalva látszik az egész folyamat leegyszerűsítve. Vegyünk példának egy összeadást, amit a WorkingCalendar.add metódus meghívása indít el, paraméter kapja a dátumot (date), amihez hozzá kell adni az időtartamot (duration). Első lépésként lekérdezzük, hogy ez az intervallum hány percet is jelent, majd az addMinutes metódus meghívásával kezdődik el a tényleges számítás. Lekérjük az aznapi munkanapot, és meghívjuk rajta az add metódust a dátummal és a még nem „elszámolt”, hátralévő percekkel (remainingMinutes). Ő végigiterál az összes aznapi munkaszakaszon (workingHours), amik az intervallum hosszát levonják a hátralévő percekből. Ezt a műveletet addig ismételjük, amíg a hátralévő percekből már nincs hátra semmi.

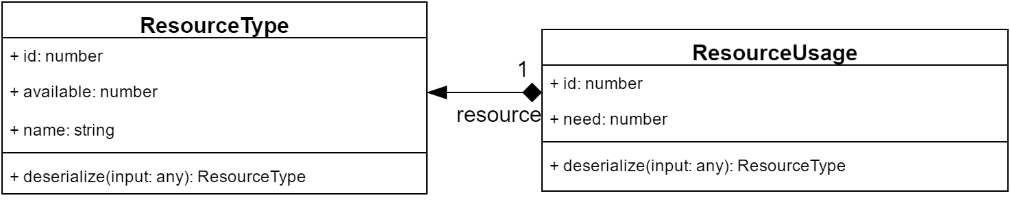
Összefoglalásképp, a fenti folyamat a projekt számára transzparens módon elvégzi azt a műveletet, ha egy dátumhoz egy időtartamot akarunk adni a munkanaptár alapján, illetve nyilvántartja a munkarendet.

### Ütemező



Fent látható az ütemező felépítése, maga a modell itt meglehetősen egyszerű, és mivel a működése az egész alkalmazásnak az alapja, ezért azt egy külön fejezetben részletezem. A Scheduler osztály végzi az ütemezést, a schedule metódusának meghívásával, és a ResourceManager osztály felelős az erőforrrások foglaltságának kezelését, és egy új feladathoz hozzárendelni az erőforrásokat.

### Erőforrások



Az erőforrások modellje szintén meglehetősen egyszerű, és a megvalósítás során az erőforrásfoglalással kapcsolatos folyamatok átkerültek az ütemező részbe. A ResourceType egy adott erőforrás, ahol megadhatjuk a nevét és azt, hogy a projekt során hány darab áll rendelkezésünkre. A ResourceUsage pedig egy feladat végrehajtásához szükséges erőforrások darabszámát és típusát tárolja

## Ütemezés, CPM

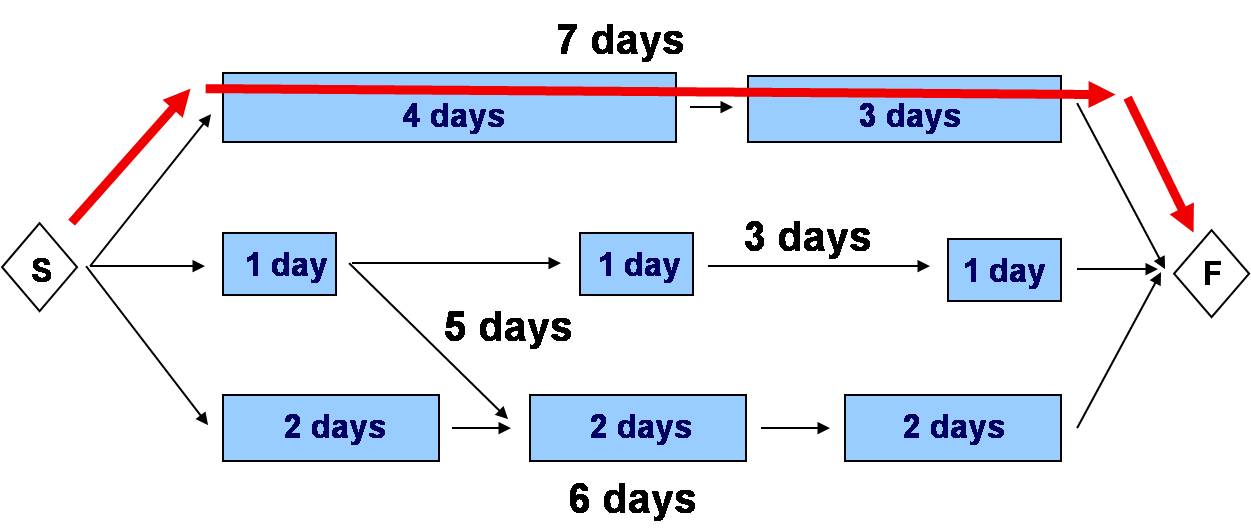
### Bevezetés

A Gantt diagram nem arról szól, hogy hogyan ütemezzük a feladatokat, hanem, hogy az eredményt hogyan jelenítsük meg a felhasználó számára. Ez az elképzelés hosszú ideig megállta a helyét, az ütemezés humán feladat volt, egészen addig, amíg a számítógépek számítási teljesítménye kellően megnövekedett ahhoz, hogy ezt automatikusan és rövid idő alatt elvégezzék.

A feladat megoldására jelenleg nem létezik olyan algoritmus, ami tökéletes eredményt adna és nem NP nehéz (<https://mialmanach.mit.bme.hu/aima/ch12s01>). A legelterjedtebb megold

### Kritikus útvonal módszer

Abban az esetben, ha előre ismerjük a feladatok végrehajtási idejét, azok erőforrás igényeit és a függőségeket, akkor első lépésként meg kell állapítanunk, hogy a feladatok lehetséges kezdési és befejezési idejét. Ezt a kritikus útvonal módszerrel tehetjük meg (CPM – Critical Path Method), amit az alábbi ábra szemléltet:



A teljes feladatsor kezdését az S (Start), míg a befejezését az F (Finish) jelzésű rombusz jelöli, közöttük kék téglalapokban láthatóak a feladatok, benne azok végrehajtási idöjével, a feladat közötti függőségeket pedig a nyilak mutatják. A nagy fekete feliratok jelölik az egyes utak (ha gráfként képzeljük el a diagramot) végrahajtási idejét.

A módszer első lépéseként megallapítjuk az előbb említett végrehajtási időket. Ha ezek megvannak, akkor kiválasztjuk a leghosszabb utat, amit kritikus útnak nevezünk (critical path, piros színnel jelölve az ábrán). Továbbiakban bármilyen ütemezőt is használjunk, a legfontosabb, hogy a kritikus úton lévő feladatokat elsőként hajtsuk végre. Könnyen belátható, hogy ez miért van így, hiszen a projekt szűk keresztmetszetét ezek a feladatok adják, ennél az útnál rövidebb idő alatt a projekt nem fejezhető be.

Amiről a CPM nem beszél, az az, hogy a többi feladatot milyen sorrendben hajtsuk végre. Erre a kérdésre a „legkevesebb tartalék először” (Least Slack Time First) módszer ad egy jó megoldást.

Az algoritmus alapját a következő heurisztika adja: végezzük el a rendelkezésre álló feladatok közül azt, amelyiknek a csúszása legnagyobb hatással van a projekt befejezési idejére, azaz melyik feladat halogatható a legkevésbé. A halogathatóság (slack time) kiszámításához minden feladathoz meg kell ismernünk, hogy mi az a legkorábbi és legkésőbbi kezdés illetve befejezés, ami még nem okoz csúszást a teljes projekt miatt. Az legkorábbi értékek a kritikus út kiszámítása közben maguktól adódnak, viszont a legkésőbbi értékek már nem adódnak ilyen egyszerűen.

Ahhoz, hogy ezek az értékek kijöjjenek, újra le kell futtatni a CPM algoritmust, most viszont visszafelé. Ez azt jelenti, hogy a projekt végén kezdjük a számítást, megfordítjuk a függőségek irányát, kezdési időpontnak beállítjuk az utolsó feladat legkorábbi befejezési időpontot (amit már ismerünk), és így indítjuk el az algoritmust.

Mikor már minden adatot ismerünk, akkor a slack time már kiszámolható a legkorábbi és legkésőbbi kezdési/befejezési időpont különbsége alapján.

### Erőforrások ütemezése

Az eddig leírt folyamat nem veszi figyelembe, hogy az erőforrások korlátosok, mert erről nem szól CPM és ahhoz kapcsolódó algoritmusok. Amit kaptunk, az az, hogy minden időpillanatban meg tudjuk mondani, hogy melyik feladat elvégzése a legjobb. Ahhoz, hogy ezt be tudjuk építeni az algoritmusba, el kell dönteni, hogy preemptív ütemezőt szeretnénk-e. Mivel, az hogy megszakítsuk egy feladat elvégzését, és egy másikkal folytassuk a sorozatot az jelent egy kis plusz időt, míg a tevékenység végzője átáll. Hogy ezt az időt beszámítsuk, nagyon megbonyolítaná az algoritmust, és ha kellően kis feladatokra bontjuk a projektet, akkor nem lenne gyorsabb egy preemptív ütemező.

Az általam választott megoldás az alábbi módon néz ki. Minden erőforráshoz annyi virtuális idővonal (továbbiakban foglaltság) kerül kialakításra, ahogy darab egység rendelkezésre áll. Ezek után kiválasztom az első olyan feladatot, aminek az összes függősége elkészült már, majd ráillesztem ezekre a foglaltságokra, mindegyik erőforrásból annyira amennyire szüksége van.



A fenti ábrán színessel vannak jelölve a már foglalt időszakok egy erőforráshoz. Ha a következő feladathoz két darab számítógépre (Computer) és egy darab fejlesztőre (Developer) van szükség, akkor őt már csak a kék időszak után tudjuk elvégezni. Ha viszont utána jön egy feladat, ami nem annyira fontos (a tartalék, vagy slack time nagyobb), de csak egy fejlesztőre van szükség az elvégzéséhez, akkor azt el is kezdhetjük rögtön, és annak ellenére, hogy nem fontos annyira, mégis előbbre kerül mint a fontosabb.

A fent leírt módszerrel már teljesen le van fedve az ütemezés problémája, mind az erőforrások mind a függőségek tekintetében.

## Megvalósítás

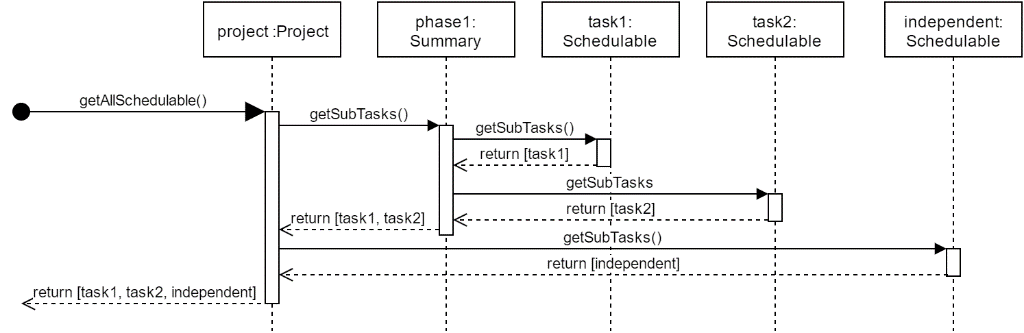
A megvalósítást egy szekvenciadiagram szemléltetné a legjobban, de az ütemezés algoritmusa sok ciklust és feltételt tartalmaz, ami elbonyolítaná a diagramot, ezért anélkül magyarázom el. Az egyszerűsítés kedvéért egy példán keresztül magyarázom el a folyamatokat, a példaprojekt következő feladatokot tartalmazza:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Név | Típus | Szülő | Időtartam | Erőforrás | Függőség |
| Phase1 | Summary | - | - | 1 fejlesztő 1 PC | - |
| Task1 | Schedulable | Phase1 | 1 nap | - | - |
| Task2 | Schedulable | Phase1 | 2 nap | 1 fejlesztő | - |
| Independent | Schedulable | - | 1 nap | - | Phase1 |

A projektben 2 fejlesztő és 2 PC áll rendelkezésre, mint erőforrás, és minden feladat elvégzése egységnyi időt vesz igénybe.

A folyamat a Project osztály schedule metódusával indul. Innentől néhány jól elkülöníthető fázis követi egymást:

* **Ütemezhető feladatok összegyűjtése**: Mivel csak a Schedulable típusú feladatok ütemezhetőek, és a projekt ezek közül csak a szülő nélkülieket ismeri, ezért ezek összegyűjtése egy külön fázis. Mivel ez az algoritmus egy egyszerű alfeladat, ezért ez szekvenciadiagrammal is szemléltethető



* **Legkorábbi kezdés és befejezés időpontjának kiszámítása:** Ennek a szükségességét már az ütemező elméleti működését leíró részben beláttuk, viszont a működését ott nem részleteztem. Ezt a részt a Project.calculateEarliestTimes metódus végzi. Első lépésként az elvégzendő feladatokat két részre osztjuk: feldolgozott, és nem feldolgozott feladatokra, kezdetben a feldolgozott üres, az összes feladat a feldolgozatlanba kerül. Amíg van feldolgozatlan feladat, addig kivesszük azt, aminek már az összes függőségét feldolgoztuk, és ennek beállítjuk a legkorábbi kezdésnek a legkésőbbi befejeződött (lag-gal együtt) függőség végét. Fontos, hogy a függőség alatt az adott feladat és az összes szülő feladat függőségeinek összegét nézzük. A legkorábbi befejezés a legkorábbi kezdés és a feladat hosszának az összege. Ezeket az összeadásokat WorkingCalendar modulon keresztül végesszük, így garantálva, hogy mindig a projekthez definiált naptár szerint zajlik a feladatok elvégzése.
* **Legkésőbbi kezdés beállítása:** Ugyanaz, mint az előző, csak itt indulásnak a legkésőbbi legkorábbi befejezést vesszük a feladatok közül. Minden művelet, ami eddig összeadás volt, az most kivonás, és a függőségek iránya is megváltozik.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Név | Legkorábbi  kezdés | Legkorábbi  befejezés | Legkésőbbi kezdés | Legkésőbbi befejezés |
| Task1 | 1 | 2 | 2 | 3 |
| Task2 | 1 | 3 | 1 | 3 |
| Independent | 3 | 4 | 3 | 4 |

Látható, hogy a kritikus út a Task2 – Independent feladatok sorozata, hiszen náluk a legkorábbi és legkésőbbi időpontok megegyeznek.

* **Ütemezés:** A fenti két lépés előkészíti az ütemezéshez a bemeneti adatokat, az ütemezést ezek után végezhetjük el, ami a Scheduler.schedule metódusban került implementálásra. Első lépésként rendezzük a már megismert halogathatóság (slack time) alapján a feladatokat. Ezek után ebben a sorrendben rendeljük hozzá az egyes feladatokhoz az erőforrásokat (emiatt hívjuk least slack time ütemezőnek). Az erőforrások hozzárendelése már ismertetésre került, de most egy példán keresztül is bemutatom.

Amíg van ütemezendő feladat, addig mindig kivesszük az első olyan feladatot a listából (ami ugye rendezett a slack time alapján), aminek az összes függőségét ütemeztük már. Megállapítom, hogy mi az az időpont, amikor legkorábban elkezdhetem a feladatot, amit úgy lehet megtenni, hogy a függőségek közül kiválasztjuk azt, ami a legkésőbb fejeződik be (lag-gal együtt).

Ezek után történik az erőforrások lefoglalása, amiért a ResourceManager.allocateTask metódus felelős. Első körben lementi az erőforrások aktuális foglaltságát, majd megpróbálja lefoglalni. Ha nem sikerül, visszaállítja a lementett erőforrások állapotát, és megkeresi a legközelebbi olyan időpontot, ami alkalmasnak tűnik a feladat végrehajtásához. Ezt addig ismételi, amíg nem sikerül lefoglalni a szükséges erőforrásokat.

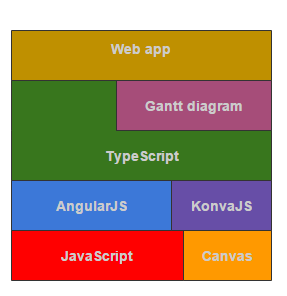
Ha egy feladatot sikerül ütemezni, akkor a szülőfeladatnak felszól, aki frissíti a saját ütemezést, így garantálva, hogy a szülők is „ütemezésre” kerülnek.

# A teljes webalkalmazás

## Bevezetés, követelmények

A feladatnak legnagyobb része az ütemező algoritmus megalkotása, és a modell kialakítása volt, viszont egy alkalmazás sikerében nem feltétlenül az algoritmusok „jósága” a fő befolyásoló tényező, hanem a felülete. Jelenleg egy jól megírt HTML alkalmazás kellően jó alapot ad, hogy olyan eredmény szülessen, ami elnyeri a felhasználók tetszését.

## Architektúra



4. ábra Architektúra

Fent látható az elkészült szoftver architektúrájának egy egyszerűsített ábrája. Az alsó rétegen a JavaScript és Canvas helyezkedik fel. Ezek olyan technológiák, amit a böngészők natív módon támogatnak, azaz semmilyen plusz könyvtár vagy beépülő modul használata nem szükséges. A KonvaJS a Canvas-t elfedő JavaScript függvénykönyvtár, míg az AngularJS magáért az egész webalkalmazás működéséért és megjelenéséért felelős eszköz.

A Web app és a Gantt diagram az általam írt komponensek, amik mind TypeScript-en kereszül érik el a fent említett komponenseket.

## Diagram rajzolás

### Lehetséges megoldások

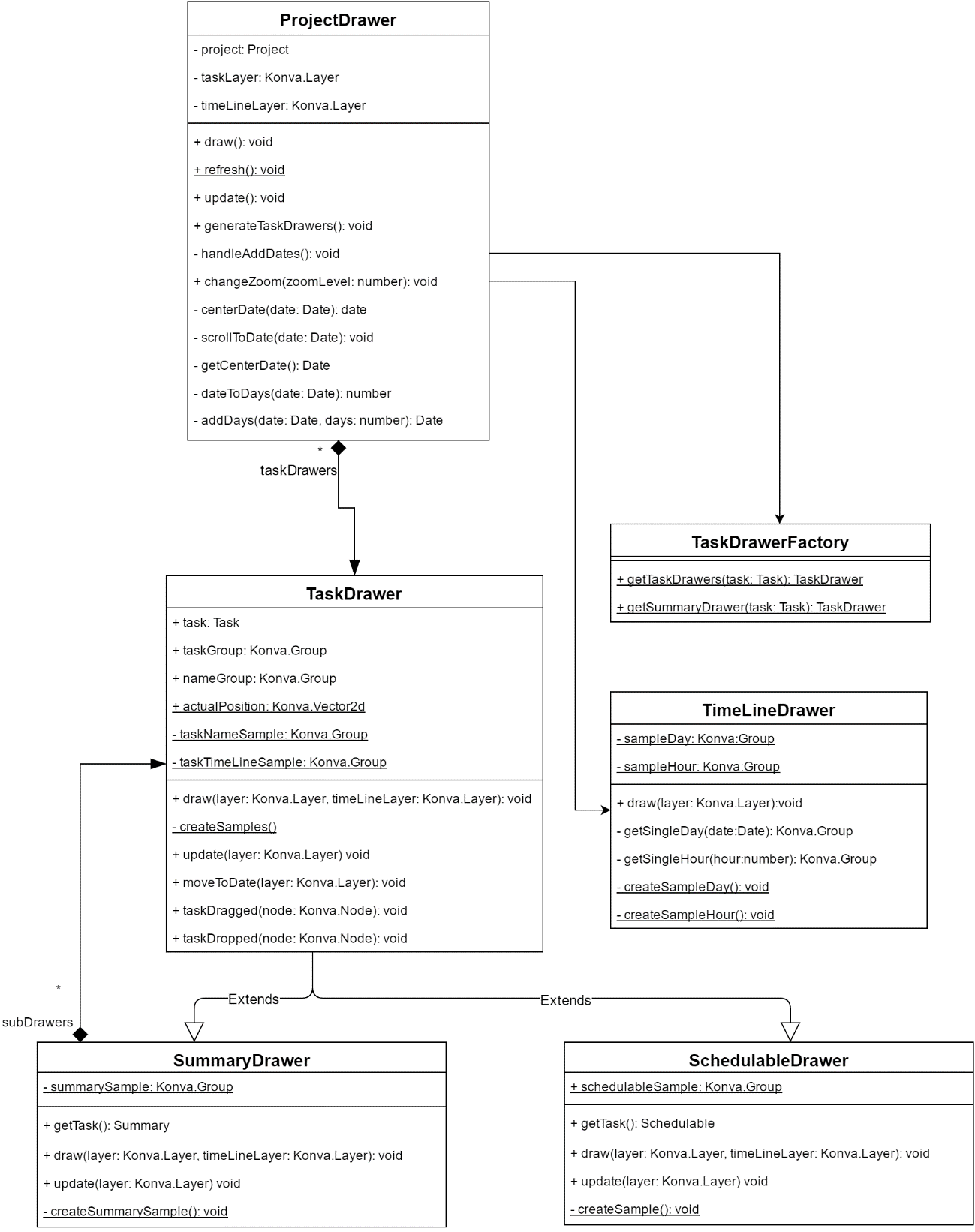
Első lépésként a diagram megjelenítéséhez használt technológiát kellett kiválasztani, amire három lehetséges alternatíva merült fel:

* **Tiszta HTML**: Mivel a technológia adott, ezért felmerült, hogy közvetlenül, minden komoly kiegészítés nélkül használjunk HTML elemeket, és azokat jelenítsük meg. Ennek az előnye az lett volna, hogy már egy nagyon kiforrott technológiáról van szó (még attól sem kell tartani, hogy esetleg a böngészők máshogy jelenítenék meg a dolgokat, az elmúlt években sok előrelépés történt ennek kiküszöbölésére), illetve az egyes megjelenített elemek DOM (Document Object Model, HTML-ből a böngésző által készített modell) elemként használhatóak, ami nagyon kényelmes. Sajnos ennek a technológiának vannak hátulütői, mégpedig az, hogy a HTML által megjelenített alakzatok száma korlátozott (konkrétan téglalap, esetleg ennek valami lekerített variánsai), és a fejlesztés végén, mikor már a megjelenés finomhangolása zajlik, esetleg ez a készlet korlátokat jelenthet.
* **SVG**: Hasonló a HTML-hez, ugyanúgy XML-szerű leíróval adható meg a megjelenítendő elemek leírása, de annál jelentősen nagyobb eszközkészlettel rendelkezik, amivel a HTML legnagyobb hátrányát kiküszöböli, ezzel alkalmassá válik a feladat elvégzésére. Ami miatt mégsem emellett döntöttem az az, hogy ez a technológia korántsem nevezhető modernnek, már régóta jelen van a piacon, és nem is kapcsolódik a HTML5-höz. Ez még önmagában nem is lenne probléma, de az alkalmazás elkészítése közben igyekeztem olyan megoldásokat választani, amik újak és úttörőek, hiszen ezek megismerése hosszabb távon több előnnyel jár.
* **Canvas**: A HTML5 bevezette a Canvas elemet, ami pixelpontos rajzolást tesz lehetővé, ezzel a megjelenített grafikák korlátlan alakzatot és formát vehetnek fel. Mivel ezt a megoldást választottam, ezért ennek bemutatását később teszem meg.

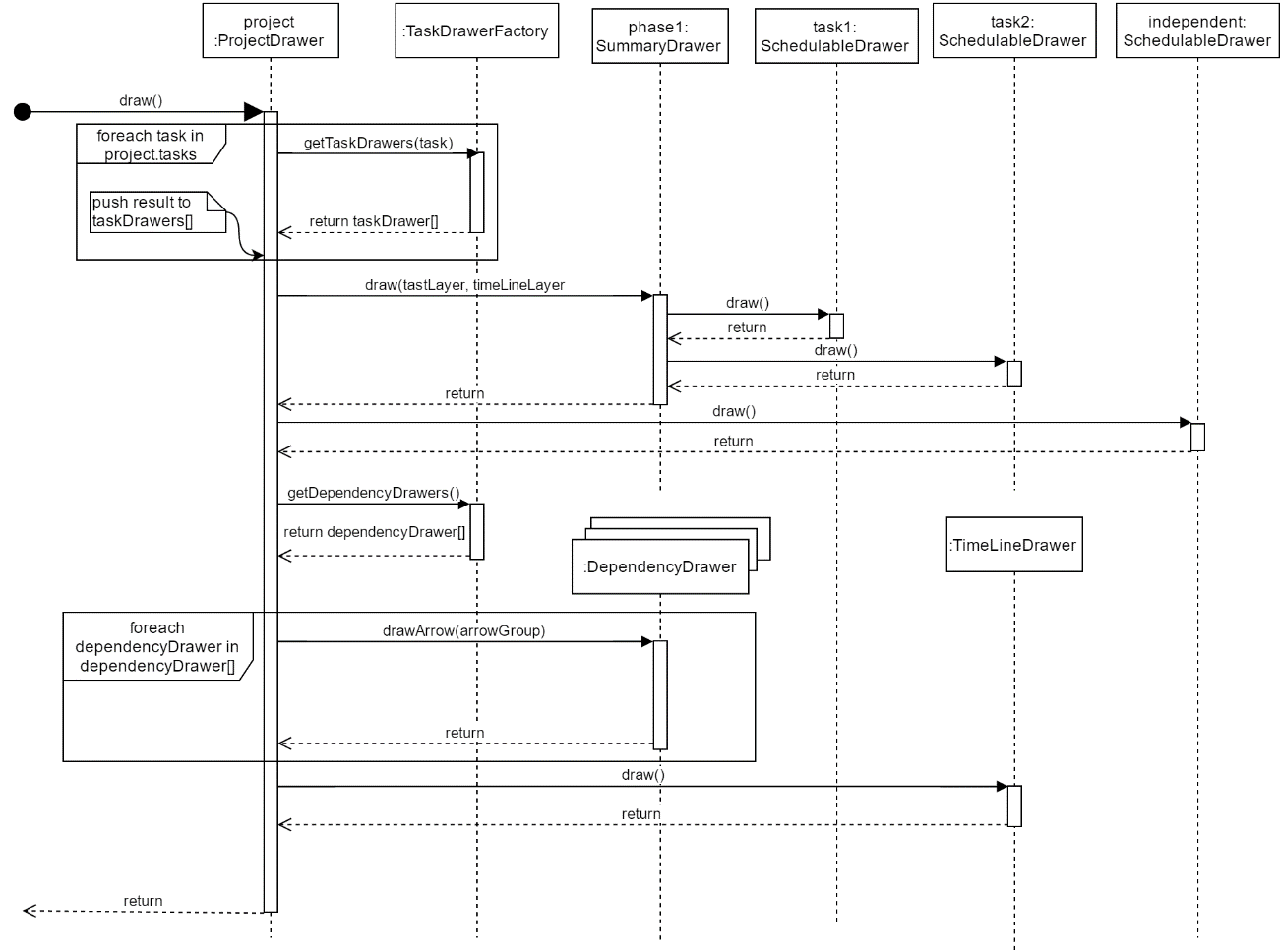
A diagram megjelenítéséhez a KonvaJS nevű eszközt használtam, ami a HTML Canvas-ra épül. Maga a Canvas egy a HTML5-ben megjelent pixelpontosan rajzolható felület, főleg egyszerűbb grafikus megjelenítésre tervezve, de WebGL-el akár komplexebb játékok is futhatnak böngészőbem

### Megvalósítás

A következő rész során bemutatom, hogy hogyan valósítottam meg a diagramm megjelenítést, ami nem tartalmazza a teljes alkalmazást, csak a szűken vett diagrammot, ami a HTML canvas-ra kerül felrajzolásra. Az alkalmazás többi részének a megjelenítését a 4.4-es pontban fejtem ki.



Fent látható a megjelenítést végző réteg osztálydiagramja. A réteg belépési pontja a ProjectDrawer osztály draw() metódusa, ami kirajzolja a diagramot. Ennek a folyamata a következőképpen történik, amihez az érthetőség kedvéért egy speciális esetet az ütemezésnél is használt példaprojektet használok.



A fenti szekvenciadiagram egy látható, ami leegyszerűsített formában tartalmazza a rajzolás folyamatát. Első lépésként a projekt összes feladatához generáltatunk egy TaskDrawer objektumot a TaskDrawerFactory segítségével, majd ezeken végigiterálva meghívjuk rajtuk a draw metódust.

A diagramon nem látszik a draw metódus felépítése, de mivel a rajzolás itt történik, így ezt a részt jobban részletezem. A SummaryDrawer és a SchedulableDrawer csak a feladat végrehajtási idejét jelző téglalap méretében és színében különbözik, így a kettőt összevonva írom le. Első futáskor minden kirajzolandó objektumból létrehozunk egyet a memóriában és azok a paramétereket, amik közösek (szín, vonalak jellege stb) beállításra kerülnek. Minden futáskor ezeket leklónozzuk, az egyedi megjelenéshez néhány paramétert módosítunk rajtuk (pl: szöveg, méret, elhelyezkedés), majd ezeket ráhelyezzük a canvas-ra.

# Fejlesztés közben felmerülő problémák

## IntervalList

Ahhoz, hogy az ismertett ütemező megfelelően tudjom működni szükség volt egy olyan osztályra, ami alkalmas intervallumok kezelésére és ütközés detektelására. Ez segít abban, hogy az erőforrásokhoz rendelt foglaltságok esetén biztosan ne alakuljon ki ütközés

# Összefoglalás

# Ábrák jegyzéke

[1. ábra Gantt diagram a Microsoft Project 2013 programban 7](#_Toc449970592)

[2. ábra Material design – anyag 14](#_Toc449970593)

[3. ábra Material Design – merészség 14](#_Toc449970594)

[4. ábra Architektúra 24](#_Toc449970595)

# Irodalomjegyzék