



MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR



TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Komponens orientált webfejlesztés Dart nyelven

Component Oriented Web Development in Dart

Szabó Gábor

II. éves mérnökinformatikus MSc hallgató

Konzulens:

Dr. habil Kovács László

egyetemi docens, intézetigazgató, dékánhelyettes

Általános Informatikai Tanszék

Miskolc, 2013

Tartalomjegyzék

Abstract	1
1. Bevezetés	1
2. A web komponensek motivációja.....	3
2.1. A HTML hiányosságai	3
2.2. Jelenlegi komponens alapú technológiák	4
2.2.1. Szerver oldali keretrendszerek.....	5
2.2.2. Kliens oldali keretrendszerek	6
2.3. Új W3C szabványok.....	8
2.3.1. Shadow DOM	8
2.3.2. Templates, Model-Driven Views.....	9
2.3.3. Custom Elements	10
2.4. A Dart nyelv	11
3. Saját módszer az adat- és kódintegráció MVC alapokon történő megvalósításához	15
3.1. Adatmegjelenítő komponensek	18
3.1.1. Carousel	18
3.1.2. Datagrid	21
3.1.3. Tagcloud	24
3.2. Dialog komponensek.....	25
3.2.1. Confirmation Dialog.....	25
3.2.2. Dialog	27
3.3. Input komponensek	28
3.3.1. Boolean Button	28
3.3.2. Rating.....	29
3.4. Multimédia komponensek	30
3.4.1. Gallery	30
3.4.2. Image Compare.....	31
3.5. Panel komponensek.....	32
3.5.1. Panel	32
3.6. Segédkomponensek	34
3.6.1. Clock.....	34
3.6.2. Growl	35
3.7. Neurális háló komponens	37
3.7.1. A mesterséges neurális hálók elmélete	37
3.7.2. A komponens működése.....	38

4. Szoftvermetrikák a komponensek komplexitásának mérésére	42
4.1. Ciklikus komplexitás	43
4.2. Halstead metrikái	45
4.3. Kohézió	46
4.4. SLOC: Source Lines of Code	47
4.5. KLCID	48
4.6. Henry és Kafura metrikája	48
4.7. Futásidő	49
4.8. Kolmogorov-komplexitás	50
4.9. Új metrika a komponensek mérésére	51
5. Az új technológia előnyei a szoftvermetrikák tükrében	54
5.1. Kódminőség	54
5.2. Fejlesztői produktivitás	55
5.3. Teljesítménynövekedés	61
6. Összefoglalás	65
Melléklet	67
Irodalomjegyzék	71

Ábrák jegyzéke

1. ábra: A date input elem belső felépítése.....	8
2. ábra: Shadow DOM.....	9
3. ábra: Model-Driven Views.....	10
4. ábra: Egyedi komponensek felhasználása	11
5. ábra: A web komponensek hierarchiája	16
6. ábra: A carousel komponens	19
7. ábra: A carousel komponens képgalériaként.....	21
8. ábra: A datagrid komponens	23
9. ábra: A tagcloud komponens.....	24
10. ábra: A confirmation dialog komponens.....	26
11. ábra: A dialog komponens.....	27
12. ábra: A boolean button komponens.....	29
13. ábra: A rating komponens	29
14. ábra: A gallery komponens	31
15. ábra: Az image compare komponens	32
16. ábra: A panel komponens.....	33
17. ábra: A clock komponens.....	34
18. ábra: A growl komponens	36
19. ábra: Előrecsatolt mesterséges neurális hálózat	37
20. ábra: A neurális háló komponens háló nézete betanítás közben	40
21. ábra: A neurális háló komponens neuron nézete betanítás közben.....	40
22. ábra: A neurális háló komponens háló nézete működés közben.....	41
23. ábra: A neurális háló komponens neuron nézete működés közben.....	41
24. ábra: A kidolgozott komponens könyvtár kohéziós gráfja.....	55
25. ábra: A szükséges Dart kód sorainak száma	57
26. ábra: A szükséges HTML kód mélysége.....	59
27. ábra: A szükséges HTML kód sorainak száma	60
28. ábra: A web komponensek cache-elt és cache nélküli betöltési ideje	64

Táblázatok jegyzéke

1. táblázat: Népszerű kliens oldali keretrendszerek	7
2. táblázat: A szükséges Dart kód sorainak száma.....	56
3. táblázat: A szükséges HTML maximális mélysége és sorainak száma	58
4. táblázat: A web komponensek compression ratio értékei százalékban mérve.....	61
5. táblázat: A komponensek betöltési idejének összehasonlítása az Ext JS és PrimeFaces keretrendszerekkel.....	62
6. táblázat: A komponensek betöltési ideje HTML5 application cache használatával	63

Kódrészletek jegyzéke

1. kódrészlet: Egy személy osztály definíciója (C#) és példányosítása (XAML).....	4
2. kódrészlet: Összeadást végző form JSF-ben	5
3. kódrészlet: Gomb definiálása jQuery UI segítségével	7
4. kódrészlet: Megtévésző Javascript kód (bal oldal) és javítása (jobb oldal)	12
5. kódrészlet: A Dart nyelv legfontosabb elemeit tartalmazó program.....	13
6. kódrészlet: A carousel komponens használata (HTML)	19
7. kódrészlet: A carousel komponens használata (Dart)	20
8. kódrészlet: A carousel komponens használata képgalériaként (HTML)	20
9. kódrészlet: A datagrid komponens használata (HTML)	22
10. kódrészlet: A datagrid komponens használata (Dart)	23
11. kódrészlet: A tagcloud komponens használata (HTML).....	24
12. kódrészlet: A tagcloud komponens használata (Dart).....	25
13. kódrészlet: A confirmation dialog komponens használata (HTML).....	25
14. kódrészlet: A confirmation dialog komponens használata (Dart).....	26
15. kódrészlet: A dialog komponens használata (HTML)	27
16. kódrészlet: A dialog komponens használata (Dart).....	28
17. kódrészlet: A boolean button komponens használata (HTML)	28
18. kódrészlet: A boolean button komponens használata (Dart).....	29
19. kódrészlet: A rating komponens használata (HTML)	29
20. kódrészlet: A rating komponens használata (Dart)	30
21. kódrészlet: A gallery komponens használata (HTML)	30
22. kódrészlet: Az image compare komponens használata (HTML)	31
23. kódrészlet: A panel komponens használata (HTML).....	33
24. kódrészlet: A panel komponens használata (Dart).....	34
25. kódrészlet: A clock komponens használata (HTML).....	35
26. kódrészlet: A growl komponens használata (HTML)	35
27. kódrészlet: A growl komponens használata (Dart)	36
28. kódrészlet: A neural network komponens használata (HTML)	39
29. kódrészlet: A programozási stílus befolyása az SLOC metrikára.....	47

Abstract

Web developers have been facing some frustrating architectural issues around HTML for years that derive from its original purpose: describing simple static documents – and not developing high-quality enterprise applications. Although HTML5 was a big step forward, it was conservative enough not to attempt to resolve these problems.

Because HTML does not provide extension points to elements, client-side frameworks like jQuery UI and Ext JS implement components purely in Javascript which makes them more difficult to use and maintain.

As a solution, Google designed a new language called Dart that is safer, faster and more elegant than Javascript, and published a family of W3C standard proposals called web components that make it possible to create and use custom components in HTML.

During my work, I've been researching the new Dart language and web components from a developer's perspective as well as performance-wise. As a conclusion, although these components aren't supported by browsers natively yet, they not only provide a more mature development style, but they can be used to encapsulate reusable code fragments and algorithms as well.

1. Bevezetés

Az utóbbi időben a web alapú fejlesztésben érdekes trend figyelhető meg: a kezdeti, szerver orientált technológiák mellett a mobil eszközök megjelenésével egyre nagyobb népszerűséget kapnak az AJAX orientált, kliens oldalon aktívabb megoldások. Ennek oka, hogy így minimalizálhatjuk a szerver és kliens közötti hálózati forgalmat, ami előnyös a kis teljesítményű eszközök számára.

Ennek a trendnek a következménye, hogy a HTML és a Javascript egyre nagyobb teret hódít a webes alkalmazások készítésében, míg más kliens oldali technológiák (pl. Java, Flash, Silverlight) háttérbe szorulnak. Mivel a HTML és Javascript nem erre a célra jött létre – hanem egyszerű, statikus dokumentumok leírására –, számos ellentmondás tapasztalható a fejlesztők igényei és a nyelvek által biztosított szolgáltatások között.

Talán a legnagyobb probléma az újrahasznosítás hiánya, hiszen a HTML nem biztosít semmilyen megoldást az öröklődésre. Mivel más technológiákban is kialakultak komponens orientált megoldások (pl. XAML, JSF, ASP.NET), a tisztán HTML alapú fejlesztésben is szükség volt hasonló elemekre. A HTML azonban nem komponens orientált, így a kliens oldali keretrendszerek, mint például a jQuery UI és az Ext JS a Javascript nyelv segítségével implementálja vizuális komponenseit.

Bár ezek használható megoldások, használatuk körülményesebb, mint más komponens orientált technológiáké, hiszen a Javascript sem nagyvállalati alkalmazások fejlesztéséhez jött létre. Sokkal egyszerűbb lenne a fejlesztők dolga, ha közvetlenül HTML-ben lehetne felhasználni a komponenseket, hiszen így modulárisabb, deklaratívabb alkalmazások jöhetnének létre, melyek karbantartása is egyszerűsödne.

Felismerve azt, hogy a natív webes nyelveket a jelenlegi igények mellett a fejlesztők kénytelenek olyan célokra használni, amelyek még nem jelentek meg a nyelvek kialakulása-kor, a Google két dolgot tett a probléma orvoslása érdekében: tervezett egy Javascriptnél lényegesen elegánsabb és biztonságosabb nyelvet, a Dartot; illetve új szabványcsaládot hozott létre, amelyek a HTML kiegészítését célozzák meg.

Az úgynevezett web komponensek lehetővé teszik, hogy az újrahasznosítható vizuális komponenseket saját komponensként definiáljuk, majd közvetlenül HTML-ben használhassuk fel őket. Ezzel a HTML-be is bekerülnek a legfontosabb objektum orientált paradigmák, mint például az egységbezárás, az információrejtés, illetve az öröklődés.

Ezek a szabványok jelenleg working draft állapotban vannak, de már most is használhatóak a modern böngészőkben, így bárki kipróbálhatja őket és visszajelezhet a tapasztalatairól annak érdekében, hogy a végső verzió a lehető legtöbb igényt kielégítse.

Ebben a dolgozatban azt vizsgálom meg, hogy milyen lehetőségei vannak jelenleg egy fejlesztőnek web komponensek létrehozására, hogyan lehet nagyvállalati alkalmazásokban is felhasználható komponenseket fejleszteni. Emellett arra a kérdésre keresem a választ, hogy milyen előnyökkel jár, ha web komponenseket használunk, illetve a ma is használható alternatív technológiákkal szemben milyen esetleges sebességnövekedést érhetünk el.

A böngésző alapú alkalmazások terjedésével nemcsak nagyvállalati szoftvereket lehet HTML-lel támogatni, így az új web komponensek használatára is sok más terület nyílik. Ennek demonstrálására egy külön komponenst hoztam létre, amely neurális hálók belső működését vizualizálja. Ez a komponens jól használható egy webes e-learning rendszerben.

2. A web komponensek motivációja

Ez a fejezet azokat a motivációkat mutatja be, amelyek a web komponensekhez vezetnek. A HTML hiányosságai után néhány olyan technológiáról olvashatunk, amelyeket jelenleg használhatunk hasonló célok elérése érdekében.

Ezek után azokkal az új szabványkezdeményezésekkel ismerkedhetünk meg, amelyek lehetővé teszik, hogy a HTML nyelvet kiegészíthessük saját elemekkel.

Végül az új Dart nyelv főbb tulajdonságairól olvashatunk, amelyek kiküszöbölik a Javascript gyengeségeit, létrehozva egy szinte páratlanul produktív nyelvet, amelyet nemcsak webes, hanem konzolos alkalmazásokban is használhatunk.

2.1. A HTML hiányosságai

A HTML [1] az 1990-es években alakult ki, célja pedig az egyszerű, statikus, interneten tárolt dokumentumok leírása volt, melyeket a böngészőkben jeleníthetünk meg.¹ A kezdeti verzió sokkal egyszerűbb volt a mainál, az olyan lehetőségek, mint például képek beágyazása és formok létrehozása csak később jelentek meg.

A HTML történetére jellemző, hogy az igényeknek megfelelően folyamatosan bővítették, foltozgatták, de az alap koncepciók megmaradtak. Az idők folyamán a klasszikus HTML mellett megjelent az XHTML [2], amely az XML népszerűvé válásának megfelelően helyesen formált és valid XML dokumentumokká alakította a weboldalakat. Bár az XHTML sokat segített a nyelv egységesítésében, megtartotta az eredeti elem- és attribútumkészletet.

A jelenlegi verzió a HTML5 [3], amely bár még nem végleges szabvány, de a böngészők nagyfokú támogatása miatt gyakorlatilag már széles körben használják. A HTML 4.01 után az ötös verzió nagyobb lépést tett a nagyvállalati webes alkalmazások frontendjének fejlesztői felé, hiszen több olyan API is megjelent benne, amely segíti a böngésző alapú alkalmazások minél felhasználóbarátabb megjelenítését.

Ennek ellenére sok fejlesztő úgy gondolja, hogy a HTML5 sem hozta el a kívánt változásokat, hiszen az elemeken ez sem változtatott, csak új elemekkel egészítette ki a meglévő készletet. Más technológiákkal összehasonlítva például furcsának és logikátlanak tűnhet, hogy az *input* elemek megjelenését és viselkedését a *type* attribútum határozza meg, és nincs nyoma az elemek közti öröklődésnek, specializációnak.

¹ A HTML nyelvi eszközkészlete jóval korlátozottabb, mint a később megjelenő XML elemei. Ráadásul kezdetben a HTML nemcsak dokumentum struktúra, hanem formátum leírására is szolgált, és csak később jelent meg a teljes mértékben megjelenítésért felelő CSS szabvány.

Emellett nagy problémát jelent az enterprise szoftverek fejlesztőinek, hogy nem definiálhatóak saját, egyedi komponensek, hiszen ez megnehezíti az újrahasznosítható, többször előforduló kódrészletek kiemelését. Hiányoznak az objektum orientált paradigmák, többek között az információrejtés és egységbezárás.

2.2. Jelenlegi komponens alapú technológiák

A HTML lassú fejlődése közben más területeken is jelentek meg hasonló leíró nyelvek, amelyek azonban célzottan az alkalmazásfejlesztőknek készültek, így a felmerülő igényeket jobban megvalósítják.

A nem webes világban a XAML [4] az, amely talán a legjobban támogatja az objektum orientált alapelveket. A XAML egy XML alapú nyelv, amelyben tetszőleges objektumokat lehet példányosítani és azok tulajdonságát beállítani. Természetesen az esetek többségében felhasználói felületek kialakításához szokták használni, kiegészítve a Microsoft erőteljes adatkötési mechanizmusával pedig jó alapot szolgáltat a népszerű WPF/Windows 8/Windows Phone/Silverlight platformokhoz.

Egy személy objektum létrehozását illusztrálja az 1. kódrészlet:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public List<string> EmailAddresses { get; set; }
}

<Person FirstName="John"
    LastName="Doe">
    <Person.EmailAddresses>
        <System:String>john.doe@google.com</System:String>
        <System:String>john.doe@gmail.com</System:String>
    </Person.EmailAddresses>
</Person>
```

1. kódrészlet: Egy személy osztály definíciója (C#) és példányosítása (XAML)

Ennek mintájára más technológiák esetén is kialakultak felhasználói felületek felépítését segítő XML alapú leíró nyelvek: a már kihalóban lévő Flash platformra írt alkalmazások felü-

letét MXML-ben, a JavaFX [5] alkalmazásokét FXML-ben, a C++ Qt alkalmazásokét pedig QML² [6] nyelven írhatjuk meg.

A web orientált technológiák között már nem ilyen egyszerű a helyzet, hiszen ezek kódját a böngészőben való megjelenítés előtt mindenképpen HTML/CSS/Javascript alakra kell konvertálni, mivel a böngészők nem tudnak mást értelmezni.

A két lehetséges megközelítés a szerver oldali és a kliens oldali keretrendszerek használata.

2.2.1. Szerver oldali keretrendszerek

A szerver oldali keretrendszerek általában szerver oldali objektumfákkal írják le a felhasználói felületet, és a felhasználói interakcióknak megfelelően folyamatosan frissítik ezt a fát.

Mivel az objektumfát tetszőleges nyelven le lehet írni, a fejlesztő magasabb absztrakciós szinten dolgozhat, a komponensek átalakítása HTML kóddá a szerver oldali futtatókörnyezet feladata.

A Java világban a legnépszerűbb szerver oldali technológia a JSF (JavaServer Faces) [7], amely a XAML-höz hasonlóan XML alapú leíró nyelvvel rendelkezik. Ezt a nyelvet rugalmasan lehet együtt használni a HTML-lel, illetve a szerver oldali objektumokkal (például EJB-kkel).

A 2. kódrészlet egy egyszerű összeadás formját tartalmazza:³

```
<h:form>
  <p:inputText value="#{calculatorBean.x}"/>
  +
  <p:inputText value="#{calculatorBean.y}"/>
  =
  <p:inputText value="#{calculatorBean.z}" readonly="true"/>
  <br/>
  <p:commandButton value="Calculate"
    actionListener="#{calculatorBean.add()}" />
</h:form>
```

2. kódrészlet: Összeadást végző form JSF-ben

² A QML nem XML, hanem JSON alapú leíró nyelv, amely gyakorlatilag a szintaktikát leszámítva ekvivalens az XML-lel.

³ A *CalculatorBean* osztály *add* metódusa a *z* attribútum értékét egyenlővé teszi az *x* és *y* attribútumok összegével.

A JSF támogatja a komponensek definiálását is, de ezt általában nem nekünk kell elvégezni: a JSF tartalmazza a HTML-beli elemeket kiterjesztő komponenseket, amelyeket könnyedén köthetünk objektumokhoz; az alap JSF keretrendszeren felül pedig bőséges kínálat mutatkozik JSF komponens könyvtárakhoz, amelyek magasabb szintű komponenseket biztosítanak számunkra. Ilyen könyvtárak például az ICEfaces [8], MyFaces [9], ADF Faces [10], vagy a sokak által legbövebbnek tartott PrimeFaces [11].

Természetesen a Microsoft technológiák között is találhatunk hasonlót: az ASP.NET Web Forms [12] szintén szerver oldali komponensfákkal dolgozik.

Az ilyen szerver oldali technológiák előnye, hogy a fejlesztő magasabb absztrakciós szinten dolgozhat, a komponensek fejlesztői pedig mindig az adott komponensre koncentrálhatnak, hiszen az oldal megjelenítésekor az oldalon elhelyezett komponensek önmaguk felelnek saját megjelenítésükért, egymástól függetlenül generálják le a szükséges HTML, CSS és Javascript kódokat.

Hátrányuk a nagy helyigény, hiszen a felhasználók által lekért oldalakat vagy meg kell őrizni szerver oldalon, vagy az oldalt kell kiegészíteni olyan rejtett, kódolt információkkal, amelyek segítenek annak szerver oldali rekonstrukciójában.

Ezt felismerve manapság elterjedőben vannak a kliens oldali keretrendszerek.

2.2.2. Kliens oldali keretrendszerek

A kliens oldali keretrendszerek alatt a különböző Javascript keretrendszereket értjük. Ezeknek több előnye is van: egyrészt nem szükséges a használatukhoz bonyolult szerver oldal, másrészt a szükséges fájlok letöltése után a működés nagy része akár teljes mértékben a kliens oldalra koncentrálódhat, felszabadítva a szerver oldali erőforrásokat.

Az okostelefonok és tabletek elterjedésével fontossá vált, hogy a hálózati forgalom minimális legyen a webes alkalmazások esetében, így megjelentek az úgynevezett egy oldalas alkalmazások (*single page application*). Ezek letöltés után kliens oldalon működnek, és a szükséges adatokat rövid webszolgáltatás hívások segítségével küldik el a szerver oldalnak.

Mivel a Javascript sem webes alkalmazások írására lett tervezve, az ilyen alkalmazások fejlesztői gyakran vesznek igénybe különféle keretrendszereket, amelyek segítenek jól strukturálni az alkalmazásokat, illetve magas szintű vizuális komponenseket biztosítanak.

A legnépszerűbb keretrendszerek közé tartozik az MVC alapú AngularJS [13] és az MVVM alapú KnockoutJS [14], amelyek biztosítják a desktop keretrendszereknél megszokott adatkötési funkciókat. Bár ezek nem tartalmazznak vizuális komponenseket, mégis sokan választják őket, hiszen jól tesztelhető, tömör keretet adnak alkalmazásainknak.

Vizuális komponens könyvtárak közé tartozik a jQuery UI [15], illetve például az Ext JS [16]. Előbbi egy könnyűsúlyú keretrendszer, míg utóbbi a nagyvállalati alkalmazásokban használt szinte összes komponenset tartalmazza, hátránya viszont, hogy használatához több nagyméretű fájl tölthető le, így a betöltése a megszokottnál hosszabb időt vehet igénybe.

Az 1. táblázat néhány népszerű kliens oldali MVC keretrendszer főbb tulajdonságait tartalmazza. [17]

	Declarative Data Binding	Rich Widgets	Easy to Integrate
Ext JS	✗	✓	✗
AngularJS	✓	✗	✓
Knockout.js	✓	✗	✓
Google Closure	✗	✓	✗
Ember.js	✓	✓	✓
Backbone.js	✗	✗	✓

1. táblázat: Népszerű kliens oldali keretrendszerek

Mivel a HTML nem tartalmaz új elemek definiálására szolgáló funkciókat, ezek a keretrendszerek tisztán Javascriptben implementálják komponenseiket, illetve felhasználásuk sem deklaratívan, hanem Javascript segítségével történik. Így nehézkes őket használni és karbantartani.

A 3. kódrészlet egy gomb definiálását mutatja jQuery UI segítségével.

```
$(function() {
    $("button")
        .button()
        .click(function(event) {
            alert("Clicked!");
            event.preventDefault();
        });
});
```

3. kódrészlet: Gomb definiálása jQuery UI segítségével

2.3. Új W3C szabványok

Az eddig említett problémák megoldása érdekében a Google új W3C szabványokat hozott létre, amelyek jelenleg working draft verzióban találhatók meg. A community széleskörű bevonásával és a gyakori változtatásokkal azt próbálják meg elérni, hogy a végtermék a lehető legtöbb igényt kielégítse majd.

Bár a böngészők még nem támogatják natívan ezeket az új funkciókat, polyfilllek [18] segítségével akár már ma is használhatjuk őket. Ezek lényege, hogy a még meg nem lévő funkciókat Javascript segítségével implementálják egészen addig, amíg a böngészők C++-ban meg nem valósítják őket – ha ez megtörténik, akkor természetesen jelentős sebességnövekedést fogunk tapasztalni.

Ebben az alfejezetben a legfontosabb szabványokat ismerhetjük meg.

2.3.1. Shadow DOM

A Shadow DOM [19] nem új koncepció, a böngészőkben már most is jelen van.

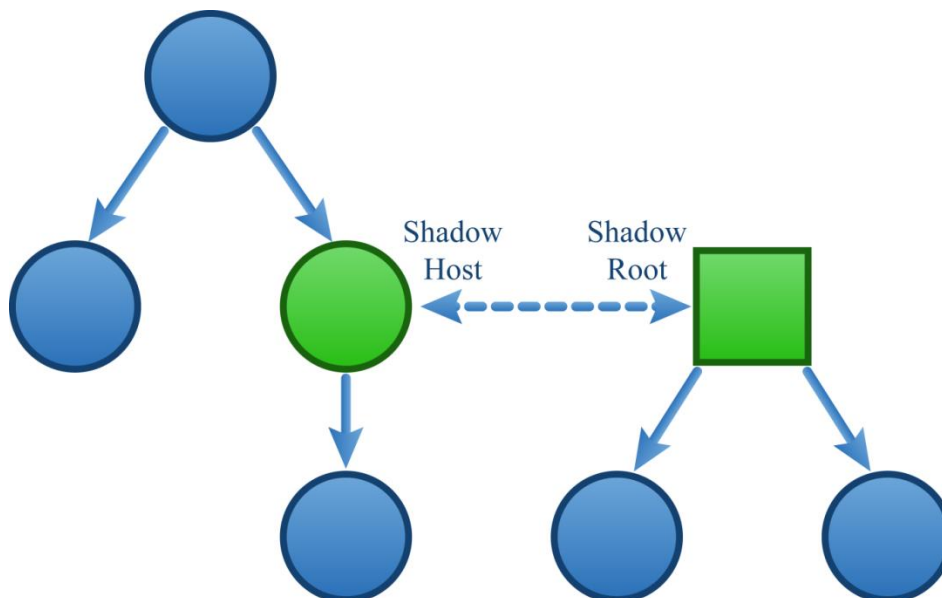
A HTML5 új elemeinek egy részét (dátumválasztó, video tag, stb.) nem hagyományos módon, C++-ban valósították meg, hanem mivel felismerték, hogy ezek felépíthetők meglévő, alacsonyabb szintű elemekből, ezekből az elemekből rakták össze őket. [20]

Ha megnézzük egy olyan oldal felépítését, amely tartalmaz egy dátumválasztó input elemet, úgy tűnhet, mintha az elem önálló entitás lenne, de ha belenézünk a belsejébe, láthatjuk, hogy több különálló *div* és *span* elemből áll, ahogyan azt az 1. ábra is illusztrálja. A technológia onnan kapta a Shadow DOM nevet, hogy a belső felépítés alapbeállítások esetén nem látszik.

```
▼ <input type="date">
  ▼ #document-fragment
    ▼ <div pseudo="-webkit-datetime-edit">
      ▼ <div pseudo="-webkit-datetime-edit-fields-wrapper">
        <span role="spinbutton" aria-valuetext="blank" aria-help="Month" aria-valuemin="1" aria-valuemax="12"
          pseudo="-webkit-datetime-edit-month-field">mm</span>
        <div pseudo="-webkit-datetime-edit-text"></div>
        <span role="spinbutton" aria-valuetext="blank" aria-help="Day" aria-valuemin="1" aria-valuemax="31"
          pseudo="-webkit-datetime-edit-day-field">dd</span>
        <div pseudo="-webkit-datetime-edit-text"></div>
        <span role="spinbutton" aria-valuetext="blank" aria-help="Year" aria-valuemin="1" aria-valuemax=
          "275760" pseudo="-webkit-datetime-edit-year-field">yyyy</span>
      </div>
    </div>
    <div pseudo="-webkit-clear-button" style="visibility: hidden;"></div>
  </div>
  <div pseudo="-webkit-calendar-picker-indicator"></div>
</input>
```

1. ábra: A date input elem belső felépítése

A Shadow DOM bevezetésével megoldható, hogy az oldalt felépítő DOM fa elemei saját belső, kívülről nem látható részfát tartalmazzanak, akár lokális CSS szabályokkal. Ez bevezeti az információrejtés elvét a HTML-be. A 2. ábra egy DOM részletet tartalmaz, ahol a kiemelt host elem egy shadow roottal rendelkezik.



2. ábra: Shadow DOM

2.3.2. Templates, Model-Driven Views

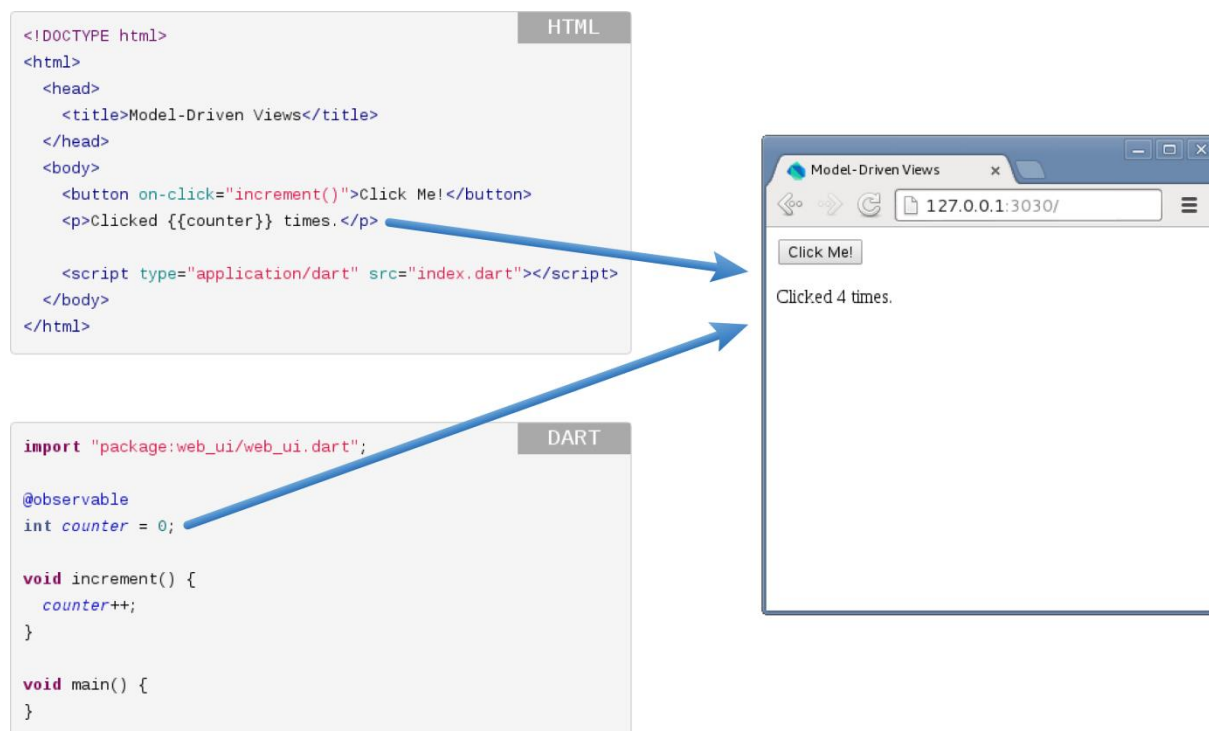
A *template* [21] egy új HTML elem, amely megoldja az újrafelhasználhatóság problémáját a HTML-ben. Ennek alapjai olyan kliens oldali Javascript keretrendszerekből származnak (pl. KnockoutJS), melyek segítik bizonyos előre definiált HTML részletek rugalmas újrahasznosítását.

A template-ek előnye, hogy a template-en belüli DOM fa nem foglal semmilyen erőforrást addig, amíg fel nem használjuk: a böngésző csak feldolgozza, parse-olja a kódot, de például nem tölt le képeket.

Ezek után klónozás útján az oldal több részén felhasználhatjuk a kódrészletet. Ez persze még nem lenne elég, hiszen így csak ugyanazt a kódrészletet lehetne másolás útján többször felhasználni.

Egy kiegészítő technológia a *model-driven views* [22], amely lehetővé teszi, hogy template-eket különböző objektumokhoz kössünk, azok tulajdonságait pedig beágyazzuk a kódba. Így megoldható, hogy az oldalon található azonos kódrészletet tartalmazó template-ek más-más adatokat jelenítsenek meg. Ha frissítjük az objektumokat, az oldal is automatikusan frissül.

A 3. ábra egy egyszerű példát mutat, ahol egy integer változót jelenítünk meg a felhasználói felületen, és a gomb megnyomásával folyamatosan növeljük annak értékét.



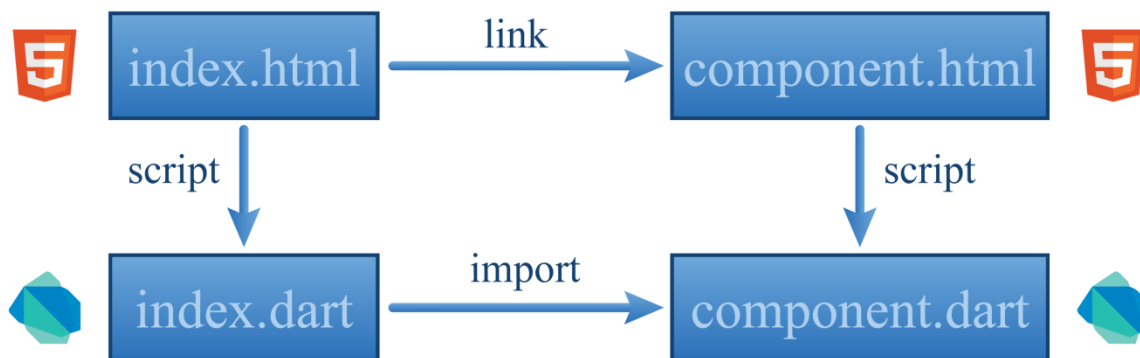
3. ábra: Model-Driven Views

2.3.3. Custom Elements

Ez a szabvány [23] írja le, hogyan lehet saját HTML elemeket definiálni, ezzel pedig megoldja az egységbezárás és az öröklődés problémáját. Az egyedi elemek tartalmazzák a megjelenítéshez szükséges HTML-t, a viselkedésüket leíró kódot, illetve opcionálisan lokális CSS szabályokat.

A felhasználáshoz a HTML oldalnak egy linket kell tartalmaznia a komponens leíró HTML fájlra. Ez a fájl a speciális *element* taggel hozza létre az új elemet, amelynek három attribútuma van: a *name* definiálja a komponens nevét, amellyel a komponens azonosítható, az *extends* attribútum jelzi, hogy milyen meglévő elem az őse, a *constructor* pedig az üzleti logikát tartalmazó osztály nevét jelöli.

Az element tagen belül egy template tartalmazza a HTML részletet, illetve opcionálisan a lokális CSS szabályokat – ez biztosítja a Shadow DOM használatát. A template tag után egy script taggel hivatkozhatunk az osztályt tartalmazó fájlra. Ha a komponens funkcionalitását is szeretnénk meghívni, akkor az oldal kódjának hivatkoznia kell a komponens kódjára. Ezt a konstrukciót illusztrálja a 4. ábra Dart nyelv esetén.



4. ábra: Egyedi komponensek felhasználása

Mivel a komponensek tartalmazhatnak gyerek elemeket, fontos, hogy ezeket az elemeket a komponens HTML kódjában hol helyezzük el. Erre a *content* tag szolgál, amelynek opcionális *select* attribútumával szűkíthetjük a megjelenítendő gyerek elemek listáját.

Ha a viselkedést implementáló osztály tartalmaz publikus tulajdonságokat, azokat a felhasználó oldalon HTML attribútumokkal állíthatjuk be, így egy olyan fejlesztői élményt kapunk, amely nagyon hasonló a szerver oldali keretrendszerekhez.

Az osztály íródhat hagyományosan Javascript nyelven, vagy a Google által tervezett új Dart nyelven, amely – mint azt a következő alfejezet is be fogja mutatni – egy elegánsabb, biztonságosabb nyelv elődjénél.

Egy egyszerű web komponens és az őt felhasználó oldal kódját megtalálhatjuk a mellékletben.

2.4. A Dart nyelv



A Javascriptnek számos hátránya és hibája van, amelyek nagy része abból származik, hogy túl nagy szabadságot ad, túl dinamikus, ezért csak nehézkesen optimalizálható, és a leírt kód nem mindig azt csinálja, amit első ránézésre várnánk tőle. Ezt jól illusztrálja a 4. kódrészlet.

A bal oldali kódrészlet – bár elsőre nem látszik – hibás, mivel a *this* kulcsszó a második hívás esetén helytelen értéket tartalmaz (az aktuális objektum helyett a *window* objektumra mutat, amely nem tartalmazza a *message* attribútumot), és ezért az alert ablakban az *undefined* értéket láthatjuk. Ezt sokan úgy oldják meg, hogy definiálnak egy *self* változót, és később azt használják, így az mindig a konstruktorban eltárolt értékre fog mutatni.

A Javascriptnek számos más hibája is van (pl. ha valaki a *var* kulcsszó nélkül deklarál egy változót, akkor az a globális térbe kerül, ami szintén nem várt és nehezen debuggolható hibákhoz vezet), ezért az utóbbi időben több nyelv is kialakult ennek orvoslására. Ezek egy

része új szintaktikát vezet be és egy fordítóval konvertálódik Javascriptre (pl. Coffeescript [24]), mások teljesen új nyelvet próbálnak meg tervezni a Javascript hibáiból okulva.

<pre>function MyObject(message) { this.message = message; this.show = function() { alert(this.message); }; } var myObject = new MyObject("msg"); myObject.show(); var showMethod = myObject.show; showMethod();</pre>	<pre>function MyObject(message) { var self = this; self.message = message; self.show = function() { alert(self.message); }; } var myObject = new MyObject("msg"); myObject.show(); var showMethod = myObject.show; showMethod();</pre>
---	---

4. kódrészlet: Megtévésző Javascript kód (bal oldal) és javítása (jobb oldal)

A Dart nyelvet [25] a Google hívta életre 2011-ben. A specifikációja [26] szerint:

“Dart is a class-based, single-inheritance, pure object-oriented programming language. Dart is optionally typed and supports reified generics.”

A Javascript nem objektum orientált, csak objektum alapú nyelv, emiatt számos olyan nyelvi konstrukciót nem tartalmaz, amelyet különböző keretrendszerek próbálnak szimulálni. Ezek a keretrendszerek egyenként jól működnek, de egy alkalmazásba integrálni őket nem mindig egyszerű.

A Dart ötvözi ezeknek a keretrendszereknek az előnyeit, illetve más népszerű nyelvek (pl. Java és C#) kedvező nyelvi elemeit is tartalmazza, kiegészítve azokat néhány ésszerűsítéssel, amelyek növelik a fejlesztői produktivitást.

Néhány magas szintű nyelvi konstrukció a teljesség igénye nélkül:

- Lexikális hatáskör: a Javascript nyelvvel ellentétben a Dartban minden hatáskör egyértelműen definiált, így nem fordulhat elő a fenti *this* kulcsszóval kapcsolatos probléma.
- Opcionális típusannotációk: a változókat használhatjuk a típusok megjelölésével vagy azok nélkül. Utóbbi esetben gyorsan készíthetünk szoftvervázlat, típusannotációkkal pedig biztonságosabb alkalmazásokat írhatunk.
- Nevesített konstruktorok: mivel a Dart dinamikus nyelv, a metódusokat nem lehet felüldefiniálni. Ez a konstruktorokra is érvényes, viszont a névvel ellátott konstruktorokkal hasonló hatás érhető el.

- Getterek és setterek: a Java és a C# nyelvénél is egyszerűbben, akár egy utasításból álló lambda kifejezésekkel definiálhatóak propertyk.
- Operator overloading: a C++ nyelvhez hasonlóan felüldefiniálhatjuk a beépített operátorokat.
- A Dart nyelv segíti a gyakran előforduló eseteket. Azokat a konstruktorokat például, amelyek csak beállítják a propertyk értékét a kapott paramétereknek megfelelően, akár egy sorral, törzs nélkül leírhatjuk.
- Cascade operator: ha több metódust is meg szeretnénk hívni egy objektumon, akkor a dupla pont operátorral ezeket egymáshoz fűzhetjük.

A Dart nyelv főbb tulajdonságait illusztrálja az 5. kódrészlet.

```
class Point {
  var _x;                                // Omitted type
  int _y;                                // Static type annotation

  Point(this._x, this._y);               // Simpler constructor syntax
  Point.zero() : _x=0, _y=0;             // Named constructor

  get x => _x;                           // Getter
  void set x(var x) => _x = x;           // Setter
  int get y => _y;                        // Typed getter
  void set y(int y) => _y = y;           // Typed setter

  Point operator +(final Point p) {      // Operator overloading
    return new Point(x + p.x, y + p.y);
  }

  String toString() => "[$x, $y]";       // String interpolation
}

void main() {
  Point p1 = new Point(1, 2);
  Point p2 = new Point.zero()
    ..x = 3
    ..y = 4;
  Point p3 = p1 + p2;
  print(p3);                             // Prints [4, 6]
}
```

5. kódrészlet: A Dart nyelv legfontosabb elemeit tartalmazó program

A Dart programok futtathatók konzolos alkalmazásként, vagy böngészőben webes alkalmazásként. Az elsődleges futtató környezet a Dart VM virtuális gép, amelynek kétféle futtatási módja van: *checked* módban minden típusannotáció kivételt vált ki hiba esetén, míg *unchecked* módban lényegesen gyorsabban, típusellenőrzések nélkül futtathatjuk a már alaposan letesztelt alkalmazásokat. A Dart VM-et beépítetten tartalmazza a Chromium böngésző fejlesztői portja, a Dartium, illetve tervezik beépítését a stabil Chrome böngészőbe is.

Annak érdekében, hogy a Dart programok Dart VM nélkül is futtathatók legyenek például olyan böngészőkben, amelyek ezt nem tartalmazzák, a *dart2js* fordítót használhatjuk a meglévő Dart kód optimalizált Javascript kóddá történő lefordításához. Ennek eredményeképpen olyan Javascript kód jön létre, amely egyes esetekben akár gyorsabb is lehet, mint ugyanazt a munkát végző, V8-ban futó Javascript kód. [27]

3. Saját módszer az adat- és kódintegráció MVC alapokon történő megvalósításához

Bár a web komponensekhez kötődő szabványkezdeményezések még nem annyira kifinomultak, hogy fejlesztői módszertanokat javasoljanak, a meglévő szabályok alapján sikerült kidolgoznom egy olyan tervezési folyamatot, amely segítségével könnyen menedzselhető kódbázist alakíthatunk ki.

Mielőtt ismertetném az új módszert, érdemes megvizsgálni, hogy a manapság használatos Javascript keretrendszerek hogyan oldják meg a komponensek finomhangolását és a minél általánosabban alkalmazható widgetek kialakítását.

Az egyik legteljesebb komponens framework az Ext JS, amely kétféle módszert javasol komponenseinek kiterjesztésére: egyrészt saját osztályhierarchiát definiál, melynek elemei egységbe zárják a komponensek megjelenését és működését, másrészt pluginek segítségével lehetővé teszi, hogy a már létező komponensek részeit felüldefiniáljuk. Ezek a pluginek szintén osztályok, amelyeket komponensekhez köthetünk, és így módosíthatjuk azok megjelenését vagy működését.

Ez a két módszer jól használható, de sajnos csak Javascript nyelven: oldalaink, komponenseink és pluginjeink megadásakor JSON objektumokat kell leírunk, amelyek elemei adat-tagok és függvények. A 2. fejezet alapján felmerül a kérdés, hogy ugyanezt a hatást miért ne érhetnénk el HTML segítségével.

A válasz nyilvánvalóan az, hogy a mai libraryk kialakulásakor még nem léteztek web komponensek, ezért tisztán Javascriptben valósították meg komponenseiket. Viszont az új technológiák alkalmazásával az adat- és kódintegráció megoldható HTML segítségével is.

A HTML a Javascripttel ellentétben nem imperatív, hanem deklaratív nyelv. Ennek előnye, hogy nem azt kell megadnunk utasításról utasításra, hogy mit kell csinálni célunk megvalósításához, hanem azt, hogy mit szeretnénk látni végeredményként. A másik előny – amely például az XML esetében is megfigyelhető –, hogy a hierarchikus felépítésnek köszönhetően áttekinthetőbb struktúrát kapunk, így valamivel könnyebb a szintaktikai elemzésen felül szemantikai vizsgálatokat is végezni.

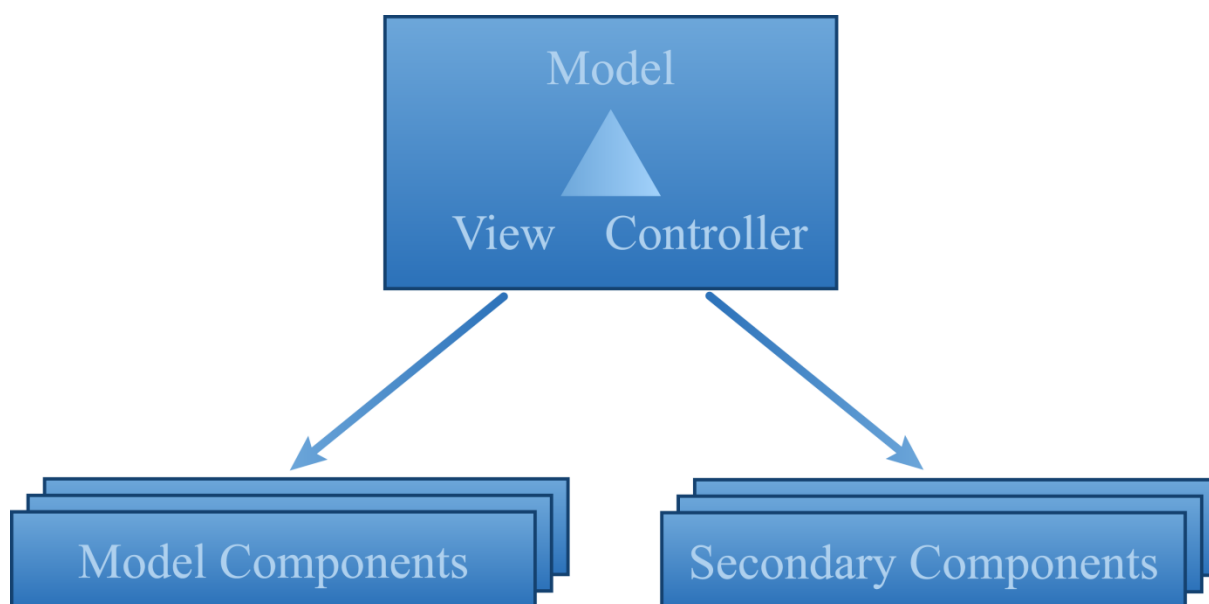
A könnyen karbantartható struktúra elérése érdekében az első lépés, hogy komponenseink írásakor az MVC tervezési mintát alkalmazzuk. Ezt a mai keretrendszerek is előszeretettel használják, hiszen segítségével a komponenseket jól szeparált, könnyebben menedzselhető osztályokként írhatjuk le.

Az MVC lényege, hogy minden komponens esetén külön osztály tartalmazza azokat az adatokat, melyek befolyásolják a komponens működését és megjelenését, külön osztály fele-

lős a komponens működéséért, illetve az előző két alapegységtől teljesen elkülönülve jelenik meg a komponens kinézetéért felelős view.

Web komponensek esetén azonban tovább is léphetünk, hiszen ha belegondolunk, komponenseink különböző szerepköröket is betölthetnek, nem mind egyenrangú. Ezeket a függőségi kapcsolatokat feltérképezve különböző kategóriákat állíthatunk fel a komponensek között.

Az 5. ábra egy általános web komponens strukturális felépítését mutatja az általam kidolgozott komponens könyvtárban.



5. ábra: A web komponensek hierarchiája

Bár a komponensek nagy része teljesen független egymástól, van néhány olyan komponens is, amely kiegészíti más komponensek működését. Ezek alapján logikailag a web komponenseknek három fő típusát különböztethetjük meg:

- Elsődleges komponensek: önálló funkcionalitással bírnak, saját célt valósítanak meg.
- Másodlagos komponensek: lehet saját megjelenésük, de funkcionalitásuk önállóan nem használható, hanem egy másik elsődleges komponens működését egészítik ki.
- Model komponensek: nem rendelkeznek saját megjelenéssel, csak adatot tárolnak, azaz a view és controller részük teljesen üres. Ezeket arra használhatjuk, hogy összetett adatokat adjunk át az elsődleges komponenseknek.

Az előző ábra alapján egy komponens struktúrája a következőképpen írható le formálisan:

```

component = (controller, model?, view?)
controller = (event*, callback+, event_handler*, property*)
property = (getter, setter?)
callback = (created | inserted)
signature(event_handler) = void _onX(XEvent event)
model = (property+)
view = template

```

A másodlagos és model komponensek ugyanúgy HTML-ben használhatóak fel, mint az elsődleges komponensek, de nem önállóan, hanem elsődleges komponensek gyerekeiként. Például ha van egy elsődleges komponensünk, amely működése során rendezést hajt végre, akkor a különböző rendezési algoritmusokat különböző másodlagos komponensekben implementálva a komponens felhasználója bármelyik változatot beinjektálhatja a komponensbe, ezzel módosítva annak viselkedését.

A hagyományos keretrendszerek is biztosítottak lehetőséget az alap funkcionalitás kiterjesztésére, de ezek esetében mindenképp imperatív kódot kellett írnia a fejlesztőnek. Web komponensek esetében a javasolt módszer egy teljesen új megközelítést nyit, hiszen a szükséges kódot egy új komponens zárja egységbe, majd ezt a komponenszt a HTML oldalon felhasználva építjük be saját megvalósításunkat a meglévő komponensekbe.

Ez nagyon hasonló egyéb technológiák XML alapú konfigurációjához, ahol a rendszer viselkedését egy XML leíró segítségével finomhangolhatjuk. Bár sokan vitatkoznak arról, hogy ez valóban a legjobb megoldás-e, az biztos, hogy ez a legrugalmasabb, hiszen az XML fájl változtatásával a rendszer automatikusan, újrafordítás nélkül alkalmazkodik igényeinkhez.

Ebből az következik, hogy a HTML-ben felhasznált konfigurációs komponensek javítják alkalmazásaink rugalmasságát, az integráció könnyebben megvalósítható és módosítható. Ha az elsődleges komponensek megfelelő számú kiterjesztési pontot biztosítanak, akkor azt mondhatjuk, hogy az integráció alacsonyabb szintre kerül, hiszen több lehetőségünk lesz belemélyülni a komponensek belső működésébe.

Természetesen nem szükséges a komponensek interfésze által kínált összes kiterjesztési ponthoz saját komponenszt írni, hiszen ez nagyban rontaná a programozhatóságot. A komponensek alkotói létrehozhatnak alapértelmezett funkcionalitású másodlagos komponenseket is, amelyeket felhasználhatunk. Ha viszont nem találunk olyat, amely megfelelő lenne, akkor saját igényeinknek megfelelően új komponenseket készíthetünk, és ezeket felhasználhatjuk az oldal leírásakor.

Ez a megközelítés teljesen új lehetőségeket nyit, hiszen az interfész alapú adat- és kód-integrációt bevezeti a HTML nyelvbe, a sok kisebb HTML fragment segítségével pedig sokkal rugalmasabban használható építőköveket hozhatunk létre, javítva alkalmazásaink karbantarthatóságát és tesztelhetőségét.

A másodlagos komponensekhez nagyon hasonlóak az úgynevezett item template-ek, amelyek már más technológiákban is elterjedtek, hiszen nagyban megkönnyítik tetszőleges adatstruktúrák vizuális megjelenítését.

WPF-ben (Windows Presentation Foundation) például tetszőleges objektumok listáját úgy jeleníthetjük meg a felhasználói felületen, hogy megadjuk az objektumok listáját és egy item template-et, amely meghatározza, hogy az objektumok mely propertyje hol és milyen formában jelenjen meg.

Bár a web komponensek és a Dart nyelv sem biztosít lehetőséget item template-ek automatikus megvalósítására, az általam készített komponensekhez ezt a mechanizmust saját kezűleg implementáltam, így a komponensek felhasználója a WPF-hez hasonló módon használhatja azokat a komponenseket, amelyek listákat jelenítenek meg. Ez nagyfokú rugalmasságot és átlátható kódot eredményez.

A következő alfejezetekben néhány olyan komponens működését és használatát mutatom be, amelyet az eddig bemutatott technológiákkal készítettem az itt ismertetett alapelvek segítségével.

A komponensek funkcionalitásának és megjelenésének alapjait a PrimeFaces keretrendszer képezi, kivéve a clock komponenst, illetve a neural network komponenst, amely azt demonstrálja, hogy az új technológiát nemcsak nagyvállalati alkalmazások építőkövének lehet használni, hanem bármilyen böngésző alapú alkalmazás alapjaként, mint például egy e-learning rendszer.

3.1. Adatmegjelenítő komponensek

Az adatmegjelenítő komponensek közé tartozik a carousel, a datagrid és a tagcloud, amelyek adatok megjelenítéséért felelősek.

3.1.1. Carousel

A carousel segítségével tetszőleges objektumokat jeleníthetünk meg egymás mellett, lapozható formában. A 6. kódrészlet illusztrálja, hogy milyen HTML kód szükséges a carousel komponens használatához.

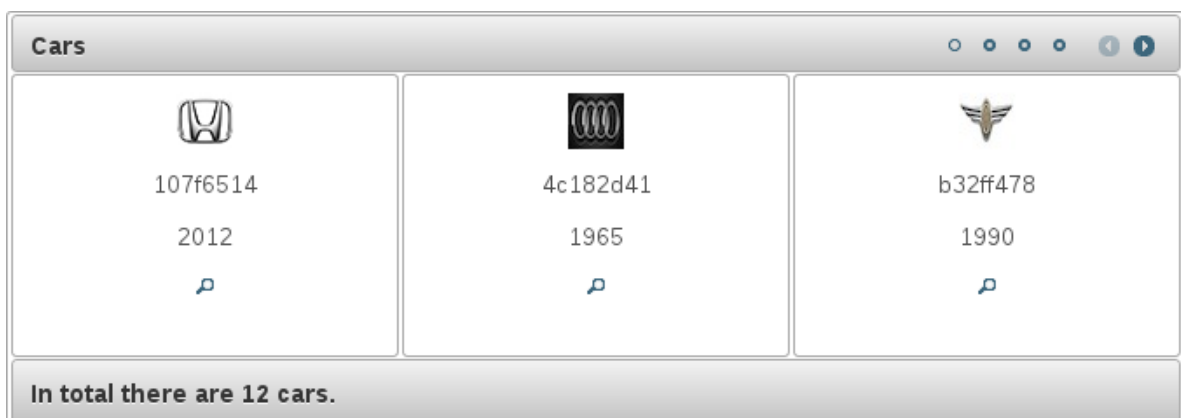
```

<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/data/carousel.html"/>
  </head>
  <body>
    <x-carousel data="/services/carousel/cars" visibleitems="3">
      <header>Cars</header>
      <template>
        <div>
          <p></p>
          <p>${model}</p>
          <p>${year}</p>
          <p>${carousel:selectButton}</p>
        </div>
      </template>
      <footer>In total there are ${carousel:totalCount} cars.</footer>
    </x-carousel>
  </body>
</html>

```

6. kódrészlet: A carousel komponens használata (HTML)

A komponenshez rendelhető fejléc és lábléc, illetve minden objektum a megadott *item template*-nek megfelelően jelenik meg, ahogyan a 6. ábra szemlélteti.



6. ábra: A carousel komponens

A 7. kódrészlet az eseménykezelő beállítását tartalmazza.

```
import "package:html_components/components/data/carousel.dart";

void setEventListeners(CarouselComponent carousel) {
  carousel.onSelected.listen((CustomEvent event) {
    var selectedItem = event.detail;
  });
}
```

7. kódrészlet: A carousel komponens használata (Dart)

A carousel komponens másik működési módja képek megjelenítésére használható: a megadott milliszekundumonként animálva megjelenik a következő kép, és beállítható, hogy az utolsó kép után az elsőre ugorjon a galéria, ahogyan a 8. kódrészletből is látszik.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/data/carousel.html"/>
  </head>
  <body>
    <x-carousel pagelinks="4" visibleitems="1"
                autoplayinterval="5000" circular="true">
      
      
      
      
    </x-carousel>
  </body>
</html>
```

8. kódrészlet: A carousel komponens használata képgalériaként (HTML)

A fenti HTML kód a 7. ábra képét jeleníti meg.



7. ábra: A carousel komponens képgalériaként

3.1.2. Datagrid

A datagrid komponens szintén objektumokat jelenít meg, de nem egymás mellett, hanem egy gridben. Attribútumként megadható, hogy az adatokat honnan vegye (például átadható egy lista, vagy egy JSON webszolgáltatás URL címe), hány sorban és hány oszlopban jelenjenek meg az objektumok, a lapozáshoz maximum hány link jelenjen meg, valamint a lapozó a komponens felett, alatt, vagy mindkét helyen jelenjen-e meg.

Az item template-et ugyanúgy lehet definiálni, mint a carousel esetén: a \$ karakter jelzi, hogy hova kell beilleszteni az aktuális objektum tulajdonságait – ahogyan a 9. kódrészletből látszik.

Így a grid olyan HTML elemekből fog állni, amelyeket az item template segítségével adunk meg.

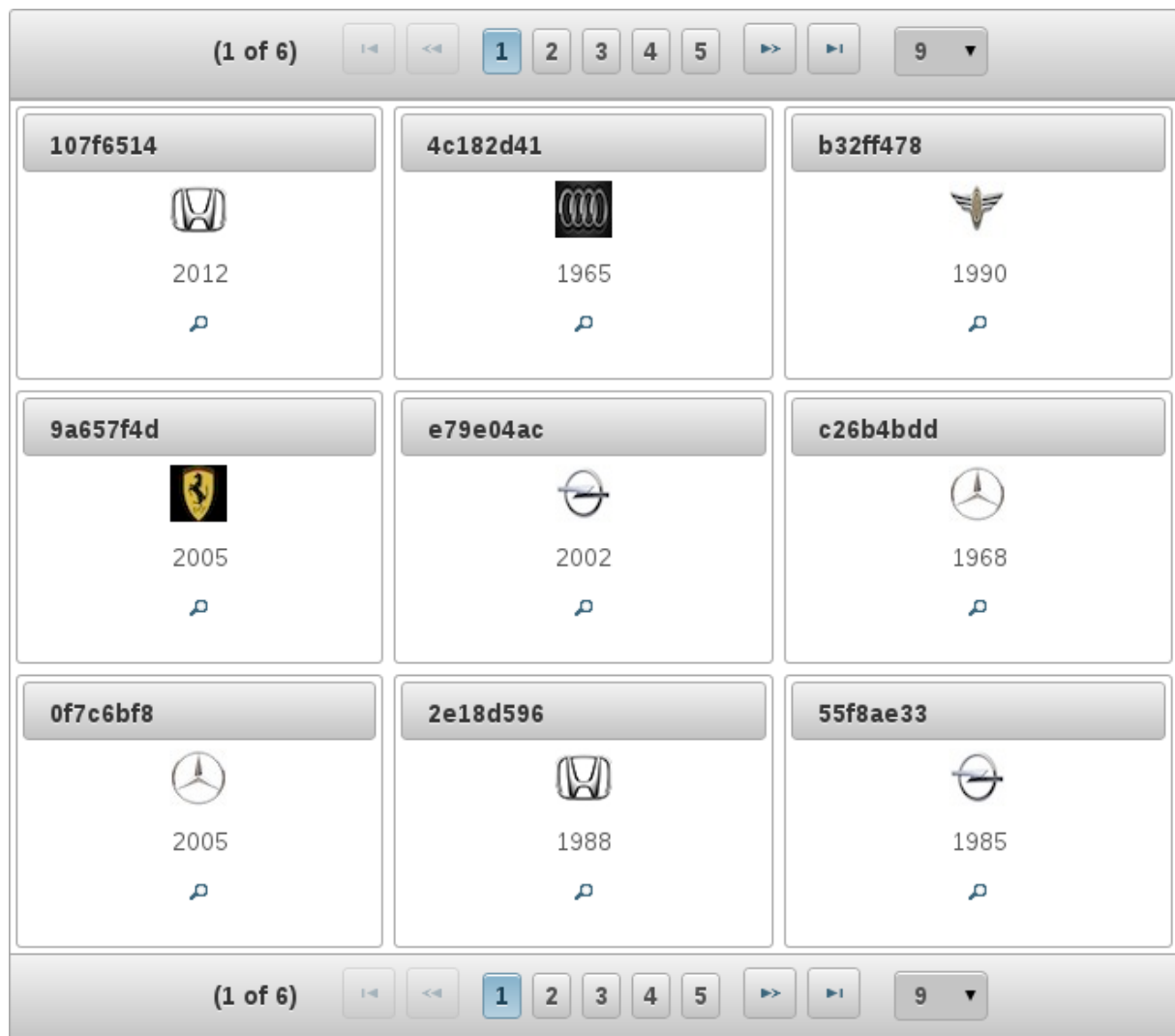
```

<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/panel/panel.html"/>
    <link rel="import"
          href="packages/html_components/components/data/datagrid.html"/>
  </head>
  <body>
    <x-datagrid data="/services/datagrid/cars" columns="2" rows="6"
                maxpagelinks="5" paginatorposition="both"
                rowsperpagetemplate="4,6,8">
      <x-panel>
        <header>${model}</header>
        <div>
          <p></p>
          <p>${year}</p>
          <p>${datagrid:selectButton}</p>
        </div>
      </x-panel>
    </x-datagrid>
  </body>
</html>

```

9. kódrészlet: A datagrid komponens használata (HTML)

A fenti kód a 8. ábra képét jeleníti meg.



8. ábra: A datagrid komponens

A 10. kódrészletből láthatjuk, hogy az eseménykezelőket hogyan lehet beállítani.

```
import "package:html_components/components/data/datagrid.dart";

void setEventListeners(DatagridComponent datagrid) {
  datagrid.onSelected.listen((CustomEvent event) {
    var selectedItem = event.detail;
  });
}
```

10. kódrészlet: A datagrid komponens használata (Dart)

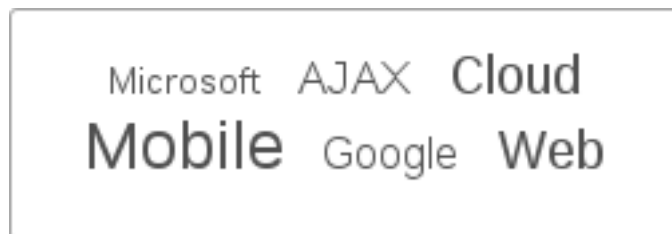
3.1.3. Tagcloud

A tagcloud komponens blogokon található címkék megjelenítésére használható. A komponens gyerekei tag elemek (ezek model komponensek), amelyek tartalmazzák az egyes címkék feliratát, erősségét és opcionálisan egy URL-t, ahogyan a 11. kódrészletből láthatjuk.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import" href="packages/html_components/components/data/tag.html"/>
    <link rel="import"
      href="packages/html_components/components/data/tagcloud.html"/>
  </head>
  <body>
    <x-tagcloud>
      <x-tag label="Microsoft" strength="1"></x-tag>
      <x-tag label="AJAX" strength="2"></x-tag>
      <x-tag label="Cloud" strength="3"></x-tag>
      <x-tag label="Mobile" strength="5"></x-tag>
      <x-tag label="Google" strength="2" url="http://www.google.com"></x-tag>
      <x-tag label="Web" strength="3"></x-tag>
    </x-tagcloud>
  </body>
</html>
```

11. kódrészlet: A tagcloud komponens használata (HTML)

A címkék ezek után az erősségüknek megfelelő méretben jelennek meg, ahogyan a 9. ábra illusztrálja. Ha rákattintunk egy címkére, akkor egy esemény váltódik ki, illetve ha be van állítva egy URL, akkor a böngésző arra a címre navigál.



9. ábra: A tagcloud komponens

A 12. kódrészletből láthatjuk, hogy a kiváltott eseményt hogyan lehet lekezelni.

```
import "package:html_components/components/data/tagcloud.dart";

void setEventListeners(TagcloudComponent tagcloud) {
  tagcloud.onSelected.listen((CustomEvent event) {
    String label = event.detail;
  });
}
```

12. kódrészlet: A tagcloud komponens használata (Dart)

3.2. Dialog komponensek

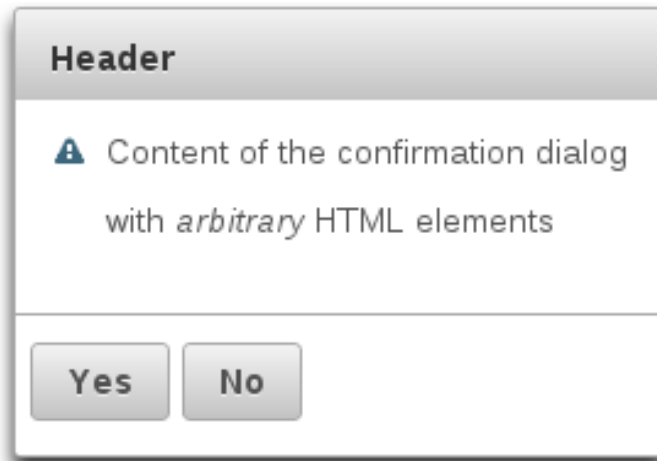
A dialog komponensek felugró ablakok megjelenítésére használhatóak.

3.2.1. Confirmation Dialog

A confirmation dialog egy kérdést és választási lehetőségeket jelenít meg. A komponensnek megadhatunk fejléceket, tartalmat, ikont, illetve gombokat. A felugró ablak lehet bezárható, illetve modal – ekkor a háttérben nem csinálhatunk semmit, amíg nem zárult be az ablak. A szükséges HTML kódot a 13. kódrészlet, a komponens képét a 10. ábra tartalmazza.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
      href="packages/html_components/components/dialog/confirmationdialog.html"/>
  </head>
  <body>
    <x-confirmationdialog modal="false" closable="false" severity="alert">
      <header>Header</header>
      <span>Content of the confirmation dialog<br/>
        with <i>arbitrary</i> HTML elements</span>
      <button on-click="onClick($event)">Yes</button>
      <button on-click="onClick($event)">No</button>
    </x-confirmationdialog>
  </body>
</html>
```

13. kódrészlet: A confirmation dialog komponens használata (HTML)



10. ábra: A confirmation dialog komponens

A gombok click eseményének lekezelését, illetve a komponens megjelenítését és eltüntetését illusztrálja a 14. kódrészlet.

```
import "package:html_components/components/dialog/confirmationdialog.dart";

void showConfirmationDialog(ConfirmationDialogComponent confirmationDialog) {
  confirmationDialog.show();
}

void hideConfirmationDialog(ConfirmationDialogComponent confirmationDialog) {
  confirmationDialog.hide();
}

void onClick(MouseEvent event) {
  String label = event.target.text;

  switch (label) {
    case "Yes":
      // The user clicked the first button
      break;

    case "No":
      // The user clicked the second button
      break;
  }
}
```

14. kódrészlet: A confirmation dialog komponens használata (Dart)

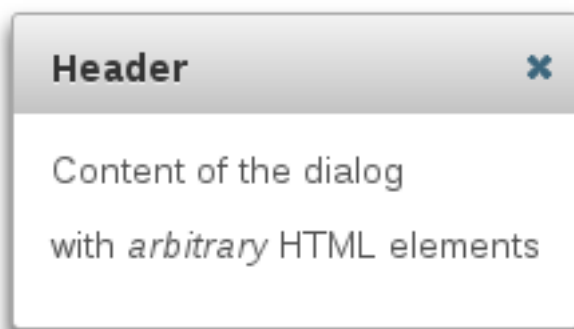
3.2.2. Dialog

A dialog komponens tetszőleges tartalmat képes megjeleníteni hasonlóan az előző komponenshez. A szükséges kód a 15. kódrészletből látható.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/dialog/dialog.html"/>
  </head>
  <body>
    <x-dialog modal="true" closable="true">
      <header>Header</header>
      <p>Content of the dialog<br/> with <i>arbitrary</i> HTML elements</p>
    </x-dialog>
  </body>
</html>
```

15. kódrészlet: A dialog komponens használata (HTML)

A komponens megjelenése (11. ábra) és metódusai (16. kódrészlet) hasonlóak az előző komponenséhez.



11. ábra: A dialog komponens

```
import "package:html_components/components/dialog/dialog.dart";

void showDialog(DialogComponent dialog) {
  dialog.show();
}

void hideDialog(DialogComponent dialog) {
  dialog.hide();
}
```

16. kódrészlet: A dialog komponens használata (Dart)

3.3. Input komponensek

Az input komponensek felhasználói adatok bevitelére szolgálnak.

3.3.1. Boolean Button

A boolean button egy olyan speciális gomb, amelynek két állapota van, így logikai értéket lehet vele megadni. A lehetséges attribútumokat a 17. kódrészlet tartalmazza.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/input/booleanbutton.html"/>
  </head>
  <body>
    <x-booleanbutton labelon="True" labeloff="False"
                     showicon="true" pressed="true">
    </x-booleanbutton>
  </body>
</html>
```

17. kódrészlet: A boolean button komponens használata (HTML)

A gomb megjelenését a 12. ábra szemlélteti.



12. ábra: A boolean button komponens

A kibocsátott eseményt a 18. kódrészlethez hasonlóan lehet lekezelni.

```
import "package:html_components/components/input/booleanbutton.dart";

void setEventListeners(BooleanButtonComponent booleanButton) {
  booleanButton.onValueChanged.listen((Event event) {
    bool newValue = booleanButton.pressed;
  });
}
```

18. kódrészlet: A boolean button komponens használata (Dart)

3.3.2. Rating

A rating komponenssel tartalmakat lehet értékelni. Meg lehet adni, hogy hány csillag jelenjen meg, illetve a komponens legyen readonly vagy disabled állapotú, valamint lehessen-e törölni a megadott értéket. A szükséges HTML kódot a 19. kódrészlet tartalmazza.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/input/rating.html"/>
  </head>
  <body>
    <x-rating stars="5" value="3" cancelable="true"
              readonly="false" disabled="false">
    </x-rating>
  </body>
</html>
```

19. kódrészlet: A rating komponens használata (HTML)

A fenti kód a 13. ábra képét jeleníti meg, a komponens értékének változását pedig a 20. kódrészlethez hasonlóan kezelhetjük le.



13. ábra: A rating komponens

```
import "package:html_components/components/input/rating.dart";

void setEventListeners(RatingComponent rating) {
  rating.onValueChanged.listen((Event event) {
    int newValue = rating.value;
  });
}
```

20. kódrészlet: A rating komponens használata (Dart)

3.4. Multimédia komponensek

A multimédia komponensek képek különféle megjelenítéséért felelősek.

3.4.1. Gallery

A gallery komponenssel képgalériát hozhatunk létre, amely automatikusan lépteti a megadott képeket, és megjeleníti a képek *alt* és *title* attribútumait. A 21. kódrészlet tartalmazza a szükséges HTML kódot, a 14. ábra pedig a megjelenő képet.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/multimedia/gallery.html"/>
  </head>
  <body>
    <x-gallery delay="5000">
      
      
      
    </x-gallery>
  </body>
</html>
```

21. kódrészlet: A gallery komponens használata (HTML)



14. ábra: A gallery komponens

3.4.2. Image Compare

Az image compare komponens két kép összehasonlítását teszi lehetővé. A két kép a 22. kódrészletből látható módon adható meg.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/multimedia/imagecompare.html"/>
  </head>
  <body>
    <x-imagecompare>
      
      
    </x-imagecompare>
  </body>
</html>
```

22. kódrészlet: Az image compare komponens használata (HTML)

Az egymásra helyezett két képet egy elválasztó vonal arrébbhúzásával hasonlíthatjuk össze (15. ábra).



15. ábra: Az image compare komponens

3.5. Panel komponensek

A panel komponensek tetszőleges tartalmak megjelenítésére használhatóak.

3.5.1. Panel

A panel komponens a megadott HTML tartalmat egy fejléccel és opcionális lábléccel jeleníti meg. A szükséges HTML kódot a 23. kódrészlet tartalmazza.

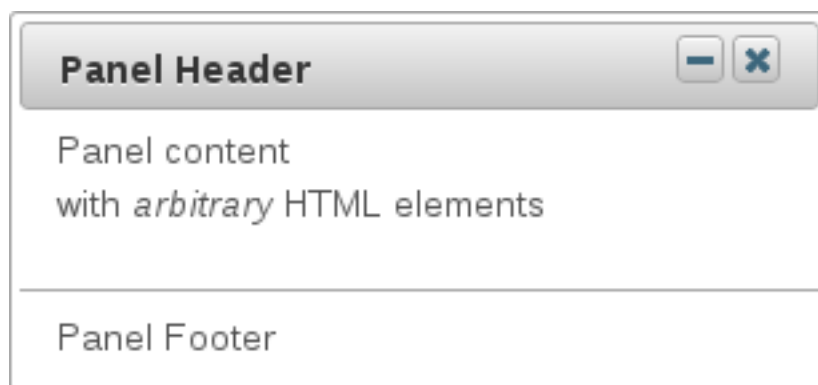
```

<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/panel/panel.html"/>
    <link rel="import"
          href="packages/html_components/components/utility/growl.html"/>
  </head>
  <body>
    <x-panel closable="true" toggleable="true">
      <header>Panel header</header>
      <p>Panel content<br/> with <i>arbitrary</i> HTML elements!</p>
      <footer>Panel footer</footer>
    </x-panel>
  </body>
</html>

```

23. kódrészlet: A panel komponens használata (HTML)

A panel fejlécében található mínuszjellel összezsukhatjuk, az X jellel pedig bezárhatjuk a panelt. A panel képét a 16. ábra szemlélteti.



16. ábra: A panel komponens

A panel eseményeit a 24. kódrészlethez hasonlóan kezelhetjük le.

```
import "package:html_components/components/panel/panel.dart";

void setEventListeners(PanelComponent panel) {
  panel.onToggled.listen((CustomEvent event) {
    if (event.detail == "HIDDEN") {
      // Panel is hidden
    }
    else if (event.detail == "VISIBLE") {
      // Panel is visible
    }
  });

  panel.onClosed.listen((Event event) {
    // Panel is closed
  });
}
```

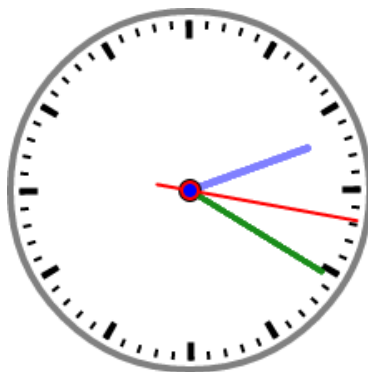
24. kódrészlet: A panel komponens használata (Dart)

3.6. Segédkomponensek

A segédkomponensek közé tartozik a clock és a growl komponens.

3.6.1. Clock

A clock komponens egy megadott méretű, SVG alapú analóg órát jelenít meg (17. ábra).



17. ábra: A clock komponens

A szükséges HTML kódot a 25. kódrészlet tartalmazza.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/utility/clock.html"/>
  </head>
  <body>
    <x-clock size="270"></x-clock>
  </body>
</html>
```

25. kódrészlet: A clock komponens használata (HTML)

3.6.2. Growl

A growl a Mac OS X operációs rendszer üzenetkezelő rendszeréhez hasonló komponens, amely különböző üzeneteket jelenít meg a böngésző jobb felső sarkában.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/html_components/components/utility/growl.html"/>
    <link rel="import"
          href="packages/html_components/components/utility/growl_message.html"/>
  </head>
  <body>
    <x-growl>
      <x-growl-message summary="Info" detail="Message detail" severity="info">
      </x-growl-message>
      <x-growl-message summary="Warn" detail="Message detail" severity="warn">
      </x-growl-message>
      <x-growl-message summary="Error" detail="Message detail" severity="error">
      </x-growl-message>
      <x-growl-message summary="Fatal" detail="Message detail" severity="fatal">
      </x-growl-message>
    </x-growl>
  </body>
</html>
```

26. kódrészlet: A growl komponens használata (HTML)

Kezdeti üzenetek a 26. kódrészlethez hasonlóan adhatók meg, az üzenetek pedig különböző ikonokkal láthatóak el, ahogyan azt a 18. ábra mutatja.



18. ábra: A growl komponens

Dinamikusan is lehet üzenetet közölni, ahogyan a 27. kódrészletből látszik.

```
import "package:html_components/components/utility/growl.dart";

void postGrowlMessage(String summary, String detail, String severity) {
  GrowlComponent.postMessage(summary, detail, severity);
}
```

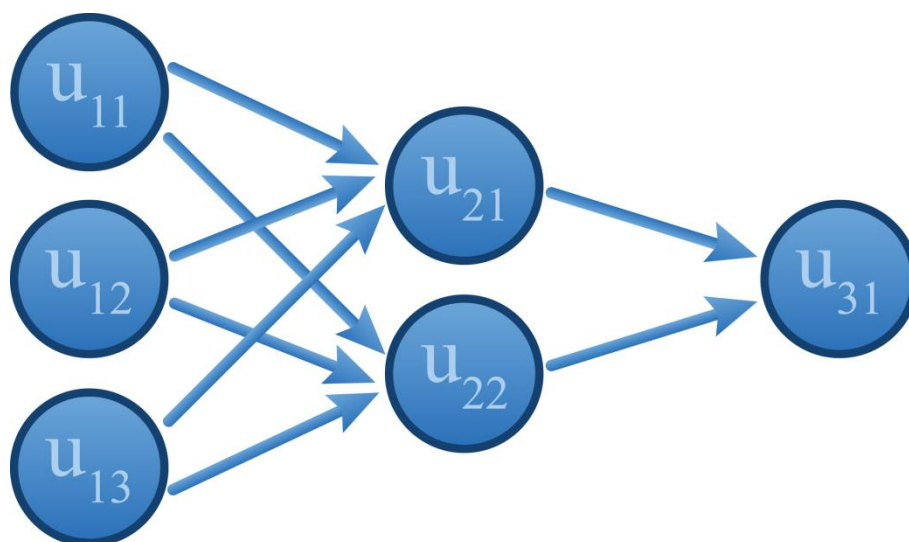
27. kódrészlet: A growl komponens használata (Dart)

3.7. Neurális háló komponens

A neurális háló komponens egy előrecsatolt (feedforward) neurális hálózat belső működését vizualizálja.

3.7.1. A mesterséges neurális hálók elmélete

A komponens egy egyszerű előrecsatolt mesterséges neurális hálót modellez, ami azt jelenti, hogy a háló neuronjai rétegekbe vannak rendezve, és működés közben az információ balról jobbra vándorol az input rétegtől az output rétegig, ahonnan leolvashatjuk a háló által szolgáltatott eredményeket. [28] Egy ilyen hálót illusztrál a 19. ábra.



19. ábra: Előrecsatolt mesterséges neurális hálózat

Az input és output rétegek között rejtett rétegek találhatóak, melyek száma és a tartalmazott neuronjainak száma tetszőleges, a problémától függ. Ezekre azért van szükség, hogy a háló „megjegyezhesse” a betanított mintákat. [29]

A szomszédos rétegek neuronjai közti kapcsolatok az agyban lévő axonokat modellezzik. Minden kapcsolathoz tartozik egy súly, amely alapvetően befolyásolja a háló működését.

Ha egy neuron kimeneti értékét szeretnénk kiszámolni, akkor szükség van még egy threshold értékre, amely a neuronra jellemző, illetve egy aktivációs függvényre, amely a kimenetet egy jól meghatározott intervallumba képezi le. A leggyakoribb aktivációs függvények közé tartozik a szigmoid függvény, melynek értékkészlete a $[0, 1]$ intervallum, valamint a tangens hiperbolikus, amely a $[-0.5, 0.5]$ intervallumba képez.

Így az i . réteg j . neuronjának kimenetét a következő képlet segítségével számíthatjuk ki:

$$o_{i,j} = \Phi \left(\left(\sum_{k=1}^{n_{i-1}} w_{i-1,k,j} \cdot o_{i-1,k} \right) - \tau \right)$$

ahol n_i jelöli az i . réteg neuronszámát, $w_{i,j,k}$ az i . réteg k . és a rákövetkező réteg j . neuronját összekötő kapcsolat súlya, τ a vizsgált neuron threshold értéke, Φ pedig az aktivációs függvénye. A fenti képlet egyszerűsíthető, ha a threshold értéket beépítjük a súlyok vektorába, az inputok vektorát pedig egy konstans 1 értékkel (bias) egészítjük ki:

$$o_{i,j} = \Phi \left(\sum_{k=1}^{n_{i-1}+1} w_{i-1,k,j} \cdot o_{i-1,k} \right)$$

ahol $w_{i-1,n_{i-1}+1,j} \equiv \tau$ és $o_{i-1,n_{i-1}+1} \equiv 1$.

Fontos kérdés, hogy a súlyokat hogyan állítsuk be ahhoz, hogy a neurális háló egy adott feladatot oldjon meg. Ehhez különböző betanítási algoritmusokat használhatunk, például az egyik leggyakoribb *backpropagation* felügyelt betanítási algoritmust. [30]

Ennek lényege, hogy a biztosított bemeneti és elvárt kimeneti értékek függvényében a súlyokat az output réteg irányából az input réteg felé haladva módosítjuk a gradiens módszert alkalmazva úgy, hogy a hiba iterációnként csökkenjen. Azt, hogy a súlyokat milyen mértékben kell korrigálni, a parciális deriváltak határozzák meg:

$$\frac{\partial E}{\partial w_{h,i,j}}$$

Itt E jelöli a hibafüggvényt, amely az elvárt és tapasztalt eredmény közti különbségből számítható ki. A parciális derivált pontos értéke az aktivációs függvénytől is függ.

Fontos megjegyezni, hogy kedvezőtlen feltételek esetén az algoritmus megrekedhet egy lokális minimumponton, így ezt az algoritmust ilyenkor célszerű más módszerekkel kombinálni. Kilépési feltételként egyrészt beállíthatjuk a maximális elfogadható hibaarányt, de a lokális minimum problémájára gondolva érdemes lehet megadni a maximális iterációk számát is.

3.7.2. A komponens működése

A komponens célja, hogy betanítás és normál működés közben megjelenítse a háló belső adatait, azaz az egyes kapcsolatok súlyát, a neuronok threshold értékét, az inputokat és outputokat, valamint az aktuális hibát.

A neurális háló megadását a 28. kódrészlet tartalmazza: egyrészt az aktivációs függvény és az adatforrás a konfigurálható attribútum, másrészt gyerek elemekként megadhatók a neuronrétegek felépítése és a backpropagation betanítási algoritmus paraméterei.

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="import"
          href="packages/neural_network/component/backpropagation.html"/>
    <link rel="import"
          href="packages/neural_network/component/neuron_layer.html"/>
    <link rel="import"
          href="packages/neural_network/component/neural_network.html"/>
  </head>
  <body>
    <x-neural-network activation="hyperbolic-tangent"
                      data="/neural-network/data">
      <x-neuron-layer neurons="5"></x-neuron-layer>
      <x-neuron-layer neurons="7"></x-neuron-layer>
      <x-neuron-layer neurons="4"></x-neuron-layer>
      <x-neuron-layer neurons="1"></x-neuron-layer>

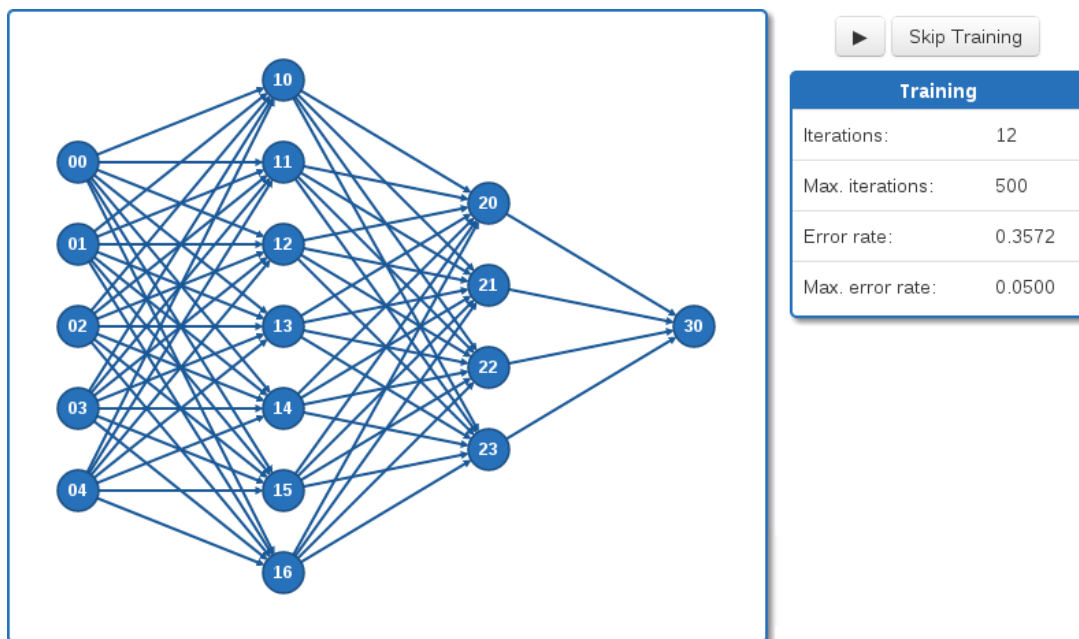
      <x-backpropagation trainfrom="0" trainto="250" maxerror="0.05"
                          maxiteration="500" learningrate="0.001" momentum="0.1">
      </x-backpropagation>
    </x-neural-network>
  </body>
</html>
```

28. kódrészlet: A neural network komponens használata (HTML)

A neuronrétegek alapján a komponens automatikusan megjeleníti a neuronokat⁴, és melléjük egy panelt, amely az aktuális adatokat tartalmazza.

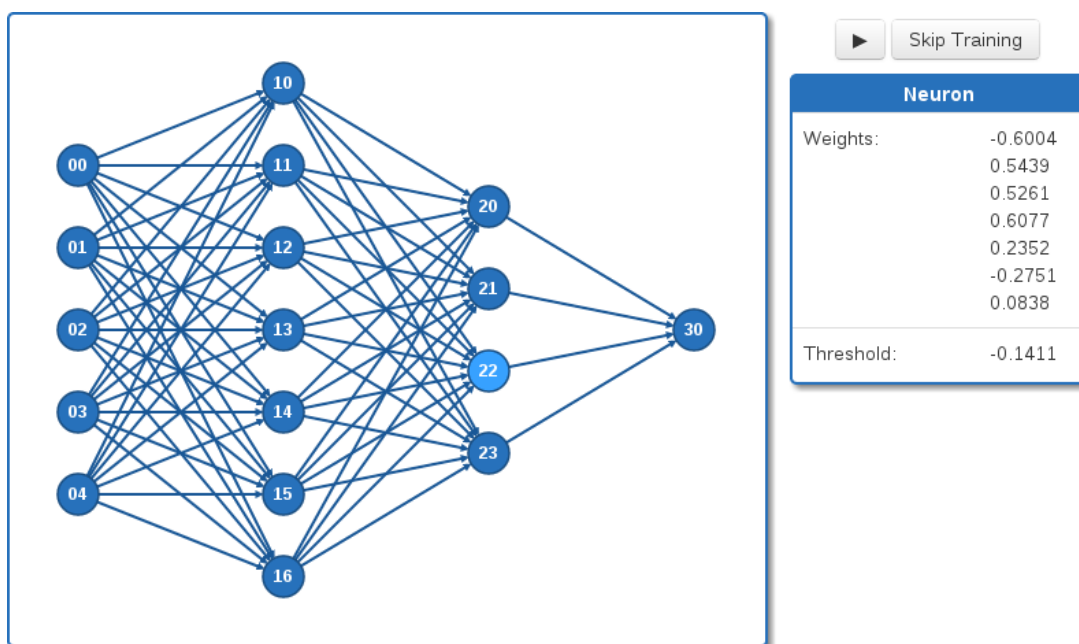
Betanítás során láthatjuk az aktuális iteráció sorszámát, a maximális iterációk számát, illetve az aktuális és a maximális hibaarányt (20. ábra). A fenti gombokkal léptethetjük, vagy átugorhatjuk a betanítást.

⁴ A neuronok drag and droppal átrendezhetők, de bármilyen struktúrát is adunk meg, a kezdeti állapot optimálisan fog megjelenni, így az átrendezésre nem feltétlenül van szükség.



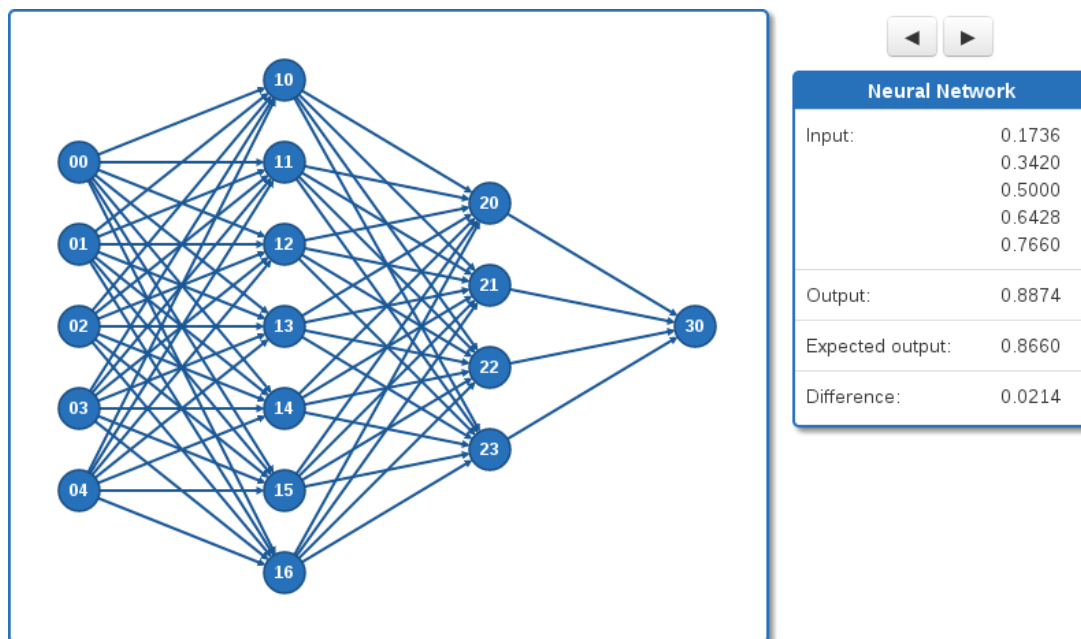
20. ábra: A neurális háló komponens háló nézete betanítás közben

Ha rákattintunk az egyik neuronra, akkor betanítás közben láthatjuk az iterációról iterációra változó súlyokat és a threshold értéket, ahogyan azt a 21. ábra illusztrálja.



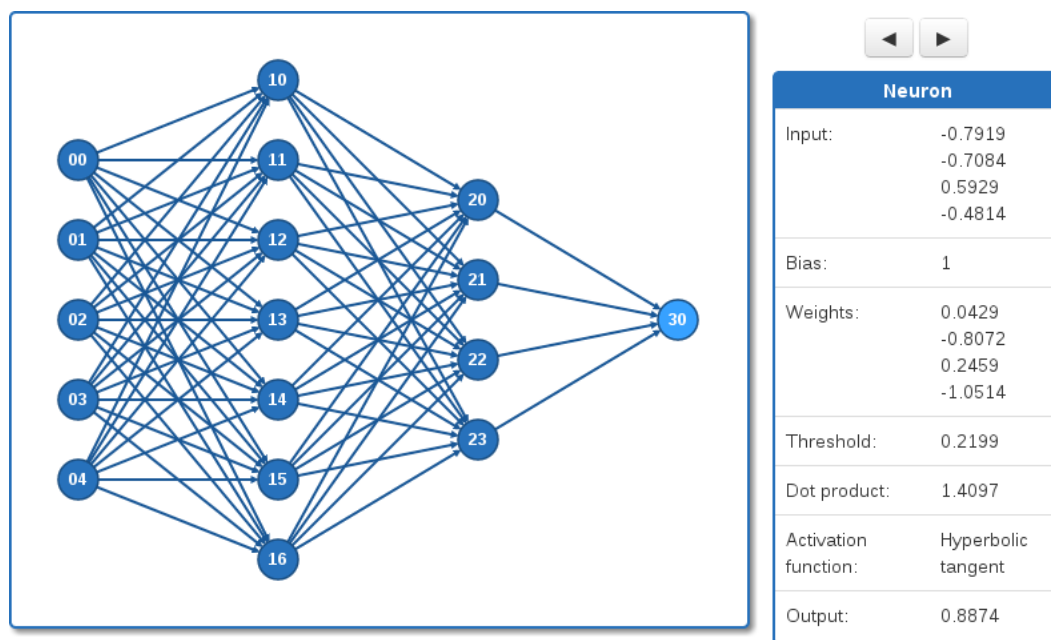
21. ábra: A neurális háló komponens neuron nézete betanítás közben

Betanítás után nyomon követhetjük a háló kimenetének alakulását. A 22. ábra jobb oldali paneljén figyelemmel kísérhetjük a háló bemenetét, kimenetét, elvárt kimenetét és a kettő közötti különbséget.



22. ábra: A neurális háló komponens háló nézete működés közben

A 23. ábra paneljének segítségével akár le is ellenőrizhetjük az egyes neuronok kimenetét: az input és a bias érték, valamint a súlyok és a threshold érték skaláris szorzatát az aktivációs függvény bemenetének adva megkaphatjuk a neuron outputját.



23. ábra: A neurális háló komponens neuron nézete működés közben

A fentiek alapján látszik, hogy a komponens egy e-learning rendszerbe elhelyezve segít a felhasználónak megérteni, hogy hogyan működnek a mesterséges intelligencia fontos részét képező mesterséges neurális hálók.

4. Szoftvermetrikák a komponensek komplexitásának mérésére

A szoftvermetrikák megjelenése a szoftverfejlesztés korai időszakára tehető. Már az 1960-as, '70-es években is kidolgozásra kerültek olyan módszerek, melyek megpróbálták objektív, mérhető mennyiségeket bevezetni annak érdekében, hogy a szoftverek minőségét értékelni tudjuk.

Mivel nincs egyetlen megoldás, amely univerzálisan leírná a programok jellegét, több módszer is kialakult, melyek más-más jellemzőkre fókuszáltak, míg másokat elhanyagoltak. Ezt a hiányosságot úgy küszöbölhetjük ki, ha minél több mérhető metrikát veszünk figyelembe, és valamilyen globális célfüggvénnyel próbáljuk meg optimalizálni szoftvereink minőségét.

Az első metrikák az Assembly és imperatív nyelvek idején jelentek meg. Bár ezeket még manapság is szokták használni, érdemes néhány ésszerű változtatást eszközölni rajtuk, hiszen azóta elterjedt az objektum orientált programozás, illetve a komponens orientált fejlesztés, amelyek magasabb absztrakciós szintet képviselnek, mint elődeik. [31]

Míg az imperatív programok alapegységét a változók és függvények alkotják, az objektum orientált szoftverek váza osztályok segítségével építhető fel, futásidőben pedig az adattagokat és tagfüggvényeket egységbe záró objektumok interakciója viszi előre a program folyamatát.

A komponensek bevezetése nem az objektum orientált paradigmák leváltását jelenti, csupán olyan még magasabb szintű alapegységekben való gondolkodást, amelyek több objektumból állnak. A komponensek komplexebb feladatot látnak el, mint egy önálló objektum, és rugalmasabban testreszabhatóak, így módosítás nélkül újrahasznosíthatóak a szoftver különböző részeiben.

A helyesen megtervezett komponensek nagyfokú függetlenséget vezetnek be a szoftver moduljai között, önállóan fejleszthetők ki és tesztelhetők. Mivel két komponens általában teljesen független egymástól, az egyikben végrehajtott módosítások nincsenek hatással a másik működésére.

A dolgozatban ismertetésre kerülő komponensek abban térnek el a klasszikus, más technológiákban fellelhető komponensektől, hogy webes környezetben használhatóak, életciklusukat így a böngésző kezeli: példányosítja őket, elhelyezi őket a DOM-ban, majd a felhasználói interakciók alapján eseményeket vált ki, amelyeket a komponens belső eseménykezelői kapnak meg.

Ezek alapján láthatjuk, hogy míg a klasszikus imperatív metrikák esetében teljes befolyásunk volt arra nézve, hogy mely függvények mikor és hogyan hívódnak meg, most csak

callback metódusokat definiálhatunk, amelyek meghívásáért maga a platform felelős. Ezért a metrikákat is más nézőpontból kell értelmezni, új szabályokat bevezetve azok mérésekor.

4.1. Ciklikus komplexitás

Az egyik klasszikus szoftvermetrika a ciklikus komplexitás, amely McCabe [32] nevéhez fűződik, és 1976-os megjelenése óta az egyik leggyakrabban említett szoftvermetrikák közé tartozik.

A ciklikus komplexitás lényege, hogy a programok reprezentálhatóak irányított gráffal, amit programgráfnak nevezünk. A strukturált programgráfok alapvetően szekvenciákból, elágazásokból és ciklusokból állnak, ezeket kötik össze az élek, amelyek a program végrehajtási menetét reprezentálják.

A programok formális vizsgálata során a gráfokat az alábbi formalizmussal írhatjuk le:

$$\begin{aligned} S(a, b) \\ E(a; b, c) \\ C(a; b) \end{aligned}$$

Itt S jelöli a szekvenciákat, azaz az utasítások egymás utáni elvégzését, E az elágazást, ahol ha az a feltétel igaz, akkor a b blokkot hajtjuk végre, különben c -t; míg C a ciklus jele, ahol a b blokkot egészen addig hajtjuk végre, amíg az a feltétel igaz.

Ezek a konstrukciók természetesen egymásba is ágyazhatóak, így egyre komplexebb programokat kaphatunk, ahol a betűket egy konkrét programnyelv utasításaival helyettesíthetjük.

A ciklikus komplexitás az ilyen programgráfok bonyolultságát méri. A legegyszerűbb számítási módja, ha megszámoljuk a gráfban lévő predikátumokat, és ezek számához hozzáadunk egyet:

$$CC(\text{algorithm}) = P + 1$$

Ez a modell az imperatív nyelvek és egyszerűbb algoritmusok esetén természetesnek hat, de megvannak a maga problémái: például egy 2000 soros program, amely 2 feltételt tartalmaz, ciklikusan kevésbé komplex, mint egy 20 soros program 5 feltétellel.

Természetesen ez egy nagyon kiélezett példa, és a szoftvertervezők felelőssége, hogy a komplex feladatokat olyan kisebb részfeladatokra bontsák, amelyek nem tartalmaznak ilyen extrém anomáliákat. Ha megfelelő szoftvervázlat alakítunk ki, akkor a ciklikus komplexitás jó képet ad a szoftver bonyolultságáról.

A komponensek esetén azonban érdemes egy szintet feljebb lépni, hiszen az imperatív programozástól magasabb absztrakciós szinten kell gondolkodnunk. Az imperatív nyelvek esetén ez a metrika függvények és procedúrák belsejének vizsgálatára használható a legjobban, komponensek esetén pedig érdemes a komponensalkotó objektumok metódusainak egymással való viszonyát szemügyre venni.

Ha a metódusokat és eseménykezelőket mint fekete dobozt vizsgáljuk, vagyis nem foglalkozunk a belső részleteikkel, felépíthetünk egy hívási gráfot belőlük. Ez a gráf szintén szekvenciákból és elágazásokból fog állni, és ha mélyebben megvizsgáljuk a hívási fát, a problémától függően ciklusokat is felfedezhetünk bennük.

Legyen $G = (V, E)$ a programgráfunk, ahol V jelöli a csúcsokat, E pedig az éleket. Ha a fenti módon akarjuk felépíteni a gráfunkat, akkor legyen $\forall m \in V$ metódus eleme a csúcsok halmazának, majd az ezek közti kapcsolatot definiáljuk úgy, hogy $(m_1, m_2) \in E$ akkor és csak akkor ha az m_1 metódusból hívás történik az m_2 metódusra.

Így egy olyan gráfot kapunk, amelyben két kapcsolatban lévő csúcs között vagy egy él fut (ez a szekvencia), vagy több – ekkor viszont nincs felső korlátja a csúcsból kiinduló élek számának, hiszen egy metódus tetszőlegesen sok más metódust hívhat meg. Annak érdekében, hogy az eredeti modellel összhangban maradjunk, a kimenő élek redukálása érdekében bevezethetünk új csúcsokat.

Például ha egy részgráfban $a \rightarrow \{b, c, d\}$ áll fenn, akkor két új csúcs bevezetésével elérhetjük, hogy a kimenő élek száma ne haladja meg a kettőt úgy, hogy:

$$\begin{array}{ll} a \rightarrow x_1, & x_1 \rightarrow b \\ a \rightarrow x_2, & x_2 \rightarrow \{c, d\} \end{array}$$

Vagyis a ciklikus bonyolultság a függvények, metódusok közti interakciók esetén is használható a komponens orientált fejlesztés esetén.

Ha most specifikusan a Dart alapú web komponenseket vizsgáljuk meg, azt tapasztalhatjuk, hogy a metódusok közti hívások optimális esetben minimálisak: a böngésző példányosítja és elhelyezi a DOM-ban a komponens, meghívva annak *created* és *inserted* callback metódusait, majd további élelciklusa során a felhasználó által kiváltott események hatására szintén a böngésző meghívja az eseménykezelő metódusokat.

Az eseménykezelők módosítják a model objektumban tárolt adatokat, melynek hatására a platform gondoskodik a megjelenített HTML tartalom frissítéséről. Ha komplex műveleteket hajtunk végre az eseménykezelőkben, akkor érdemes az újrahasznosítható kódrészeket privát metódusokba kiemelni, és ezeket meghívni – így bevezetjük a hívási gráfba a szekvenciát.

Elágazás akkor jelenik meg, ha például a komponens belső állapota alapján az eseménykezelő különböző segédmetódusokat hív meg. A ciklusokat a fenti fekete doboz modell

segítségével nem azonosíthatjuk, ehhez a metóduson belüli ciklusokat kellene megvizsgálni, és megfigyelni, hogy a ciklusmagban hívunk-e valamilyen metódust.

A fenti modell alapján elmondható, hogy ha a feltételes hívások számát minimalizáljuk, akkor az így felépített programgráf túlnyomó része $S(event_handler, utility_method)$ alakú lesz, ami a ciklikus bonyolultság szempontjából előnyös.

4.2. Halstead metrikái

Egy másik klasszikus módszer a szoftverek komplexitásának mérése a Halstead-féle metrikák alkalmazása, amelyeket Maurice Halstead vezetett le 1977-ben.

A McCabe-féle ciklikus komplexitással ellentétben Halstead metrikái nem egyetlen mérőszámot adnak eredményül, hiszen így a módszer a komplexitásnak csak egy kis szeletét vizsgálhatná. Ehelyett Halstead több mint egy tucat képletet alkotott meg annak érdekében, hogy ezeket egymással szembeállítva teljesebb képet kapjunk.

A formulák kiszámításához először meg kell határozni négy mennyiséget a szoftverrel kapcsolatban:

- η_1 : az egymástól különböző operátorok száma
- η_2 : az egymástól különböző operandusok száma
- N_1 : az operátorok teljes száma
- N_2 : az operandusok teljes száma

Ezekből a következő formulákat számíthatjuk ki: [33]

Vocabulary:	$\eta = \eta_1 + \eta_2$
Size:	$N = N_1 + N_2$
Calculated size:	$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
Volume:	$V = N \cdot \log_2 \eta$
Difficulty:	$D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$
Effort:	$E = D \cdot V$
Testing time:	$T = \frac{E}{18}$

Bár Halstead munkája úttörő volt korában, formulái nem használhatóak egzakt mérőszámokként, inkább egy adott szoftver változása során adnak képet a komplexitás változásáról, így elsősorban összehasonlítható mérőszámokként kezelhetők. [34]

Egy másik probléma a metrikákkal kapcsolatban, hogy mindenfajta számítás előtt kénytelenek vagyunk összeszámolni a programban szereplő operátorokat és operandusokat, ami nem egyszerű feladat. Ráadásul változtatás esetén ezek száma jelentősen módosulhat.

4.3. Kohézió

A kohézió vizsgálatokor azt kell tanulmányozni, hogy a szoftver egyes komponensei, építőelemei mennyire függenek egymástól. [35]

A modern nagyvállalati alkalmazások esetén fontos szempont az architektúra tervezése esetén, hogy az üzleti logika minél egyszerűbben elkészíthető és tesztelhető, egymástól független komponensekben legyenek megírva – ez az úgynevezett *loosely coupled architecture*, amely könnyű menedzselhetőséget tesz lehetővé.

A komponens orientált programozás célja pontosan az ilyen független komponensek megtervezése, implementálása és tesztelése. Web komponensek esetén a függőségnek több fokát is bevezethetjük, ahogyan azt már a 3 fejezetben is olvashattuk.

Az optimális függetlenségi szint az, amikor két komponens teljesen független egymástól, egy osztályuk sem tartalmaz referenciát a másik komponenshez tartozó objektumra.

Annak érdekében azonban, hogy a weblapon felhasznált komponens rugalmasan paraméterezhető legyen, szükség van az elsődleges komponensek mellett olyan komponensekre is, amelyek csak konfigurációs adatot tartalmaznak, ezek a model komponensek, hiszen nem tartalmaznak sem viewt, sem controllert.

Ezekkel a model komponensekkel adhatunk meg a HTML attribútumoknál komplexebb adatot az elsődleges komponensek számára, azok gyerek elemeiként. Mivel a model komponensek nem értelmezhetőek és nem használhatóak önmagukban, nem jelent igazi problémát a köztük és az elsődleges komponensek közti függőség.

A függőség következő szintje az, amikor két komponens között funkcionalitásbeli függőség figyelhető meg. Ilyenre akkor lehet szükség, ha több komponens használja ugyanazt a funkcionalitást, vagy a komponenseknek van közös view részletük. Például ilyen lehet egy paginator lapozó komponens, amely szintén nem használható önmagában, de megjelenhet több elsődleges komponensen belül is, transzparense a komponens felhasználója számára.

Ezek a másodlagos komponensek sem sértik a komponens orientált fejlesztés alapelveit, hiszen azzal, hogy az újrahasznosítható kódrészleteket kiemelik, csökkentik a kódismétlések számát. Így ha ezeket módosítani kell, akkor csak egy segédkomponensen kell változtatni, az elsődleges komponensek továbbra is működőképesek maradhatnak.

A két elsődleges komponens közti teljes függőség a legrosszabb függőségi típus, hiszen ilyenkor éppen a komponens orientált programozási alapelvek ellen vétünk. Ahogyan láthat-

tuk, ez a függőség nem jelenik meg az általam kidolgozott komponens könyvtárban, hiszen a különböző célokat megvalósító komponensek közti függőség nem indokolt.

A komponensek közti függőségeket formálisan egy $M_{n \times n} = [m_{ij}]$ mátrixszal is leírhatjuk, ahol a mátrix elemei jelzik, hogy az i . és j . komponens között milyen típusú a függőség, például 0 ha függetlenek, 1 ha a j . komponens model komponens, 2 ha másodlagos komponens és 3 ha teljesen összefüggők.

4.4. SLOC: Source Lines of Code

Bármely programozási nyelvben alapvető mérőszám a forráskód hossza, azaz az SLOC (Source Lines of Code). [36]

Bár ez a metrika bármely program esetén kiszámítható, nem számít objektív mérőszámnak, hiszen a programozási nyelvek, technológiák és fejlesztők programozási stílusa szerint ugyanaz a program más és más forráskód hosszt mutathat.

Erre a problémára jó példa az alábbi, C++ nyelvű kódrészlet:

<pre>for (int i = 0; i < 10; ++i) { std::cout << i << std::endl; }</pre>	<pre>// Printing numbers from 0 to 9 for (int i = 0; i < 10; ++i) { std::cout << i << std::endl; }</pre>
---	---

29. kódrészlet: A programozási stílus befolyása az SLOC metrikára

Emiatt szokásos LLOC metrikáról beszélni, amely a logikai LOC rövidítése, és azt méri, hogy a forráskód milyen hosszú lenne, ha eltávolítanánk a kommentárokat és a felesleges sortöréseket. A fenti példa esetén az LLOC érték mindkét esetben 2.

Web komponensek esetén nemcsak a Dart kód sorainak hosszát mérhetjük, hanem a HTML kód hosszát, illetve annak mélységét is. Ez utóbbi mérőszám azért fontos, mert HTML esetén a szokásos kódolási konvenció szerint a gyerekelemeket tartalmazó elem kezdő- és zárótagját külön sorba írjuk, feleslegesen növelve a sorok számát az olvashatóság érdekében. Ezzel szemben a HTML fa mélysége jól tükrözi a HTML komplexitását.

4.5. KLCID

Klemola és Rilling [37] alkotta meg a KLCID metrikát 2003-ban, amely az előző LOC metrikát finomítja azzal, hogy bevezeti a változók számának vizsgálatát is a formulájába.

A metrika számítása nem egyszerű feladat, [38] hiszen első lépésként meg kell határozni az egyedi sorok számát a programban. Ha két sor azonos típusú változókat tartalmaz azonos operátorokkal, akkor a két sor ekvivalensnek tekinthető.

Mivel elhagyjuk az ismétlődő sorokat, így például egy ötszörösen egymásba ágyazott for ciklus csak egyetlen ciklusként fog szerepelni a számításokban, így ez a metrika nem veszi figyelembe a vezérlési szerkezetek egymásba ágyazását.

Jelöljük az egyedi sorok halmazát U -val, és legyen ID helyettesítő szimbólum a programban felhasznált azonosítókra. Az U halmaz elemei sorok, jelöljük ezeket l -lel. Ekkor meghatározhatunk egy olyan mérőszámot, amely egy adott sorban szereplő változók számát jelöli:

$$count_{ID}(l) = |\{ID \in l\}|$$

Ezt a mennyiséget felhasználva a KLCID metrika nem más, mint a program egyedi soraiban szereplő azonosítók számának összege, osztva azon egyedi sorok számával, amelyek tartalmazznak változót:

$$KLCID = \frac{\sum_{l \in U} count_{ID}(l)}{|\{l \in U \mid count_{ID}(l) > 0\}|}$$

Ahogy a formulából is látszik, a KLCID nem veszi figyelembe a szoftvereket felépítő vezérlési struktúrákat, kizárólag a sorok és változók számával foglalkozik. Így nem meglepő, hogy nem mindig ad jó képet egy-egy szoftver komplexitásáról, de más módszerekkel kiegészítve árnyalja azok hiányosságait.

4.6. Henry és Kafura metrikája

Henry és Kafura egy olyan mérőszámot vezetett be, amely az előző metrikával ellentétben könnyen számítható, és mivel függvények helyett modulokban gondolkodik, közelebb áll a komponens orientált programozáshoz.

Számításához először két értéket kell meghatároznunk a modulra:

- fan-in: a modulba történő behívások száma
- fan-out: a modul által hívott egyéb modulok száma

Ez a két mérőszám a mai modern fejlesztőkörnyezetekben is elérhető, bár nem így hívják őket. Általában lehetőségünk van egy adott függvény felhasználási helyeit megkeresni, illetve néhány integrált fejlesztőeszköz képes felrajzolni forráskód alapján egy függvény hívási fáját is.

Ez a két adat valódi jelentéssel is bír, nemcsak absztrakt fogalmak: például ha egy függvény vagy modul fan-in értéke 0, akkor az azt jelenti, hogy a programon belül egyáltalán nem használjuk. Az ilyen kódrészek keresése segíthet a *dead code* eltávolításában.

Miután meghatároztuk a modul fan-in és fan-out értékét, kiszámíthatjuk a strukturális komplexitást. A képlet felírásához jelöljük a fan-in értéket fi -vel, míg a fan-out értéket fo -val: [39]

$$C_m = (fi \cdot fo)^2$$

Ez a komplexitás érték a modul más modulokhoz viszonyított szerepét írja le. Ha a forráskód hosszát is figyelembe akarjuk venni, akkor felhasználhatjuk az előzőleg már definiált SLOC értéket: [40]

$$C'_m = SLOC \cdot (C_m) = SLOC \cdot (fi \cdot fo)^2$$

A metrika előnye, hogy a vezérlési szerkezetekre koncentrálnak mérőszámokkal ellentétben az adat orientált alkalmazások komplexitásáról is képet ad, illetve már a tervezés során is figyelembe vehető, így a fejlesztés korai szakaszában is módosíthatjuk szoftvereink struktúráját annak érdekében, hogy a Henry–Kafura-féle mérőszám szempontjából megfelelő értékeket kapjunk a végtermékre.

4.7. Futásidő

A futásidő talán a legfontosabb nemfunkcionális követelmény egy szoftverrel szemben, amelyet a legnehezebb megvalósítani, hiszen gyakran a futtatókörnyezet, a fordító, illetve a platform is befolyással van rá.

A C/C++ és egyéb natív nyelvek esetén van a legnagyobb befolyásunk a futásidőre – főleg ha a kritikus részeket Assembly nyelven implementáljuk –, míg az olyan virtuális géppel dolgozó nyelveknél, mint a Java és a C# gyakran még hosszas optimalizálással sem tudjuk megközelíteni a natív nyelvek sebességét.

Az azonban egyértelmű, hogy a nem megfelelően megírt szoftver lassabb, mint az, amelyet már a tervezéstől kezdve úgy készítünk, hogy figyelembe vesszük a teljesítmény követelményeket.

Web komponensek esetén a böngésző biztosítja a futtató környezetet, így a kódminőségén kívül a böngészőben lévő Javascript vagy Dart virtuális gép az, amely befolyásolja szoftvereinek futásidejét.

Webes alkalmazásoknál az AJAX technológia segítségével elérhető, hogy az oldal betöltése után ne legyen szükség újratöltésre. Az általam kidolgozott komponensek támogatják az AJAX-ot, így a futásidő, illetve a felhasználói élmény javításához elsősorban a betöltési időt kell optimalizálni.

4.8. Kolmogorov-komplexitás

A szoftverek forráskódját nemcsak a strukturált programozás alapelemeiből felépített szoftvervázként lehet vizsgálni, hanem egy szöveggént, stringként is. Például a fordítóprogramok is ebből a megközelítésből indulnak ki, majd ebből alakítanak ki egy számukra optimális absztrakt szintaxis fát.

A szoftverek komplexitásmérésének is van olyan megközelítése, amely a programot mint stringet vizsgálja, és azt méri, hogy ezt a stringet milyen bonyolult leírással lehet előállítani. Ez a leírás lehet természetes szövegű, vagy megfogalmazhatjuk valamilyen programnyelven. [41]

Tekintsünk például két 8 bites sorozatot:

01010101

01101001

Az első esetben egyszerűen megadható az a szabály, amely ezt a sorozatot előállítja, például Python nyelven:

```
print "01" * 4
```

A második eset leírása viszont már bonyolultabb, főleg ha a minimális programot szeretnénk előállítani valamilyen szabályszerűséget felismerve. Ilyenkor egyszerűbb lehet a stringet egy olyan programmal előállítani, amely a stringet kiprinteli.

A Kolmogorov-komplexitás annak a minimális, L nyelven írt programnak a hossza, amely egy adott w kimenetet produkál: [42]

$$K_L(w) = \min_{p \in L \wedge p()=w} |p|$$

Egy másik megközelítés, ha olyan programot keresünk, amely egy adott input függvényében állítja elő a kívánt kimenetet – ez a feltételes Kolmogorov-komplexitás:

$$K_L(w | x) = \min_{p \in L \wedge p(x)=w} |p|$$

A Kolmogorov-komplexitás értékének könnyű felső korlátot adni, hiszen ha belegondolunk, a program hossza csak konstans értékkel haladhatja meg a kívánt output hosszát:

$$K_L(w) \leq |w| + c_1$$

$$K_L(w | x) \leq |w| + c_2$$

Ez könnyen belátható, hiszen az a program, amely a kívánt outputot írja ki, mindig megfelelő lesz, ez a program pedig csak a kimenet és az adott nyelv kiírató utasításának hosszától függ.

A nyelvek különbözőségét ki is küszöbölhetjük, ha közös nyelvet választunk. Mivel a Kolmogorov-komplexitás elméleti jellegű, választhatjuk az univerzális Turing-gépet mint közös nyelvet, így kereshetjük a legegyszerűbb olyan univerzális Turing-gépet, amelynek outputja a keresett string.

A Kolmogorov-komplexitás azonban nem kiszámítható függvény, azaz nem létezik olyan Turing-gép, amelynek egy adott stringet inputként adva az inputot elfogadja és outputja pontosan a bemenet Kolmogorov-komplexitása lesz.

4.9. Új metrika a komponensek mérésére

Az eddigi, klasszikus metrikák után megpróbálhatunk olyan mérőszámot megalkotni, amely közelebb áll a komponens orientált, magas szintű fejlesztéshez, és azt fejezi ki, hogy a komponens orientált technológiák alkalmazása milyen mértékben teszi könnyebbé a fejlesztési folyamatot.

A komponens orientált fejlesztés előnye, hogy csökkenti a kódbázis egységei közti függőséget, javítva ezzel a szoftver karbantarthatóságát. Ugyanezt a célt szolgálják a programozás során használt magas szintű eszközök és keretrendszerek.

Egy szoftver fejlesztése során érdemes lehet megvizsgálni, hogy a magas szintű technológiák és programozási paradigmák mennyiben javítják a menedzselhetőséget. Az új metrikát tömörítési aránynak (*compression ratio*) is nevezhetjük, hiszen azt méri, hogy ezek a technológiák milyen mértékben csökkentik a fejlesztő által írt kód hosszát.

Az alapötlet a strukturált programozás alapelveiből származik, hiszen az imperatív nyelvek megjelenésének egyik nagy előnye az volt, hogy az új, strukturált vezérlési szerkeze-

tek (if, for, while, stb.) jelentősen javították a programok futási szálának követését. Vegyünk példaként egy egyszerű for ciklust:

```
for (int i = 1; i <= 5; i++) {  
    System.out.println(i);  
}
```

Ezt a kódrészletet olvasva mindenki láthatja, hogy a program elszámol 1-től 5-ig, és minden iterációban kiírja az aktuális értéket. A Kolmogorov-komplexitáshoz hasonlóan a ciklust tekinthetjük egy három sorból álló stringnek.

Ha megvizsgáljuk, hogy mi történik végrehajtáskor, egy jóval terjengősebb kódot kapunk. Ha nem lenne ciklus, illetve nem használnánk más – például Assembly – kódot a ciklus megszervezéséhez, akkor a következő kódot kellene leírnunk ahhoz, hogy ugyanazt a hatást érjük el:

```
int i = 1;  
System.out.println(i);  
i++;  
System.out.println(i);  
i++;  
System.out.println(i);  
i++;  
System.out.println(i);  
i++;  
System.out.println(i);  
i++;
```

Ha a két kód viszonyát vizsgáljuk, megállapíthatjuk, hogy a for ciklus mint strukturált programozási vezérlési szerkezet tömörítési aránya ebben az esetben 27%, hiszen míg vele 3 sort kellett leírnunk, addig nélküle ugyanazt a hatást 11 sorral érhetjük el.

Mivel a web komponensek a strukturált, imperatív programozásnál jóval magasabb szintet képviselnek, nem kell ezen a szinten megrekednünk. Ahogyan már az SLOC metrikánál is olvashattuk, komponenseinket a HTML kód hosszával és mélységével, illetve a Dart kód hosszával jellemezhetjük.

Ebből kiindulva a compression ratio metrikát úgy számíthatjuk ki, hogy elosztjuk a komponensekkel megvalósított szoftverekben lévő metrikákat a komponensek nélkül megvalósított szoftverek metrikáival. Így egy olyan arányt kaphatunk, amely megmutatja, hogy az eddigi kód hány százalékára van szükség ugyanannak a feladatnak a megvalósításához az új technológia segítségével.

Ez a metrika általánosítható is, hiszen tetszőleges technológiai szinten kiszámítható. Ahogy már többször utaltam rá, az idők során a programozási nyelveknek egyre magasabb szintjei jelentek meg: kezdetben kizárólag gépi kódban lehetett fejleszteni, majd megjelent a még mindig alacsony szintű Assembly nyelv, amelyet az imperatív, strukturált nyelvek váltottak fel, majd elterjedtek az objektum orientált és komponens alapú technológiák.

Ezeket a szinteket rangsorolhatjuk is, például legyen a gépi kód 0-s szintű, az Assembly 1-es, és így tovább.

Ha van egy tetszőleges M metrikánk és egy P programunk, akkor a programot megvalósíthatjuk tetszőleges szinten, és az adott szintre kiszámíthatjuk a metrika pontos értékét. Jelölje ezt az értéket az i . szintű program esetén $M_i(P)$.

Ekkor az $(i + 1)$ -edik szint tömörítési arányát a következőképpen számíthatjuk ki:

$$CR_{i+1}(M, P) = \frac{M_{i+1}(P)}{M_i(P)}$$

Természetesen a fenti szintek képlékenyek, hiszen programozási paradigmán belül is alakulhatnak ki olyan elvek és technológiák, amelyek javítják a programozhatóságot, illetve idővel megjelenhetnek újabb és újabb módszerek.

A következő fejezetben a már bemutatott komponensek esetén fogom meghatározni a compression ratio értékét.

5. Az új technológia előnyei a szoftvermetrikák tükrében

Az új technológia számos előnnyel rendelkezik. Ebben a fejezetben főleg a web komponenseket tartalmazó alkalmazások kódminőségére, a szükséges kód hosszára és a betöltési sebességre fogok koncentrálni, hiszen ezek a legfontosabb előnyök, amelyek a hagyományos technológiákkal szemben felsorakoztathatók.

5.1. Kódminőség

A Dart tervezési céljaiból adódóan ugyanannyi energiabefektetéssel sokkal biztonságosabb kód írható, mint Javascriptben. A 2.4. alfejezetben már említett problémák nem lépnek fel, így az eredmény ugyanazt a célt biztonságosabban valósítja meg, kevesebb a nehezen debuggolható nem várt hiba.

Ezen kívül a Dart kód a Dart VM-ben gyorsabban hajtható végre, hiszen a nyelv maga úgy lett tervezve, hogy a végrehajtás könnyebben optimalizálható legyen, mint a Javascript kódoké.

A Google által végzett benchmarkok alapján [27] a Dart VM-ben végrehajtott kód minden esetben gyorsabb, mint a V8-as motorban végrehajtott Javascript kód. Egy teszt esetében (DeltaBlue) már a dart2js fordító által előállított kód is megelőzte a V8-at.

A komponensek előnye, hogy a sokszor felhasznált funkcionalitást, vizuális elemeket, vagy akár nem vizuális funkciókat ki lehessen emelni egy közös helyre, így ezeket a komponenseket könnyebb karbantartani.

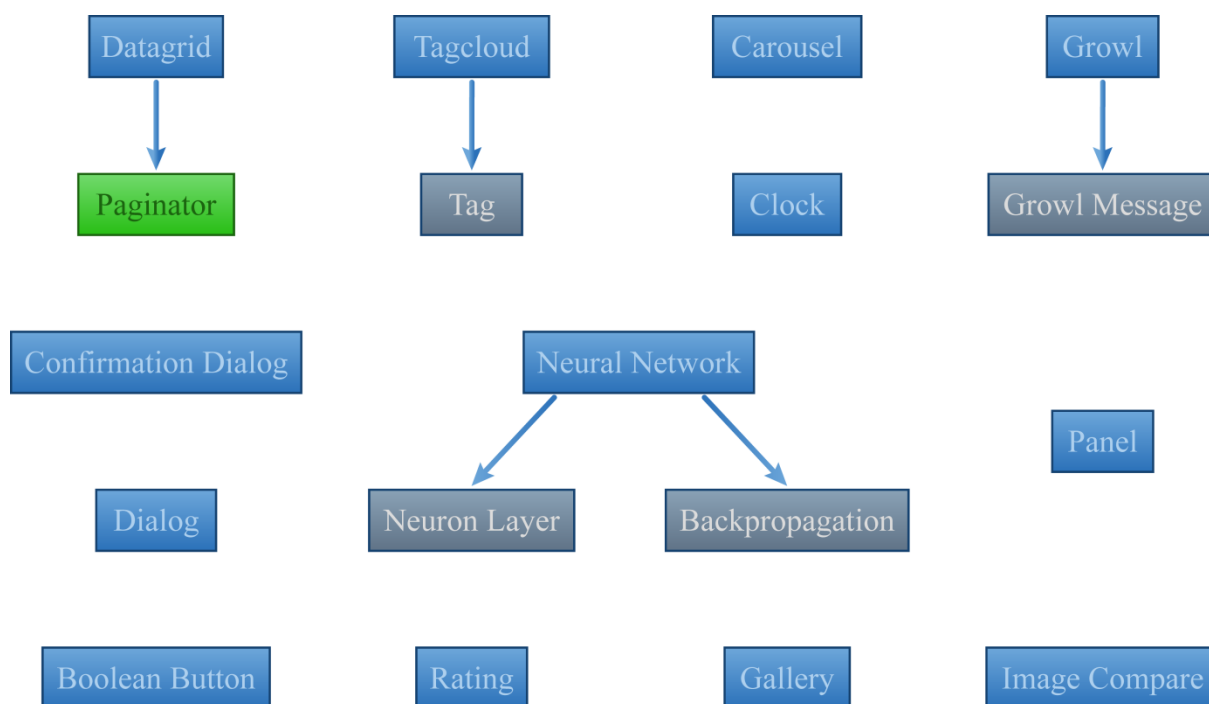
Következésképpen a végtermékül kapott alkalmazások átláthatóbbak, könnyebben menedzselhetők, modulárisak lesznek, amely összességében javítja a kódminőséget.

Ahogy a 4.3. fejezetben olvashattuk, a kódminőség mérésére szolgáló egyik olyan gyakori szoftvermetrika, amely a komponens orientált programozással is összhangban van, a kohézió.

Ennek vizsgálatához felírhatjuk a kohéziós mátrixot, amely jelen esetben – a model és másodlagos komponenseket beleértve – egy 18x18-as mátrix lenne. A mátrix nagy része 0-ból állna, hiszen a legtöbb komponens nem függ egymástól, és elvértve látnánk model és másodlagos komponensekre utaló 1-eseket és 2-eseket.

Mivel a komponens könyvtár függőségi viszonyai szerencsére igen kedvezőek, kohéziós mátrix helyett a 24. ábra kohéziós gráfja szemlélteti a függőségeket. Az ábrán az elsődle-

ges komponensek kék, a model komponensek szürke, míg a másodlagos komponensek zöld színnel láthatók.



24. ábra: A kidolgozott komponens könyvtár kohéziós gráfja

5.2. Fejlesztői produktivitás

A 3. fejezet kódrészleteiből egyértelműen látszik, hogy az új technológiák növelik a fejlesztői produktivitást is, hiszen a komponensek használatához kevesebb HTML és Dart (vagy Javascript) kódot kell írni.

A kód sorainak száma (SLOC és LLOC) csökken, illetve a HTML összetettsége, egymásba ágyazottságának mértéke, vagyis a DOM fa mélysége is jelentősen mérséklődik.

A 2. táblázat tartalmazza a szükséges Dart kód sorainak számát az előző komponensek esetén. Az első oszlop úgy jeleníti meg az adatokat, mintha komponensek nélkül valósítottuk volna meg az összes funkciót, míg a második oszlop a komponensekkel való interakcióhoz szükséges kódsorok számát tartalmazza.

Látható, hogy a várakozásoknak megfelelően a második oszlop adatai jóval alacsonyabban, mint az elsőé, gyakorlatilag konstans nagyságúak, hiszen csak attól függenek, hogy az adott komponensnek hány eseménye van, és mennyire bonyolult az a munka, amelyet az eseménykezelőkben végzünk.

	Komponens nélkül	Komponenssel
Boolean Button	80	12
Carousel	501	12
Clock	81	0
Confirmation Dialog	140	11
Datagrid	382	12
Dialog	108	11
Gallery	136	0
Growl	280	12
Image Compare	104	0
Panel	183	15
Rating	143	12
Tagcloud	70	12
Neural Network	1564	0

2. táblázat: A szükséges Dart kód sorainak száma

A clock komponens esetén fontos megemlíteni, hogy a web komponensek jelenlegi verziója nem támogatja a HTML-be ágyazott SVG kódot, így ez is a Dart kódban van leírva. Ha ez nem így lenne, az 32 sorral rövidítené a komponensen belüli Dart kód hosszát.

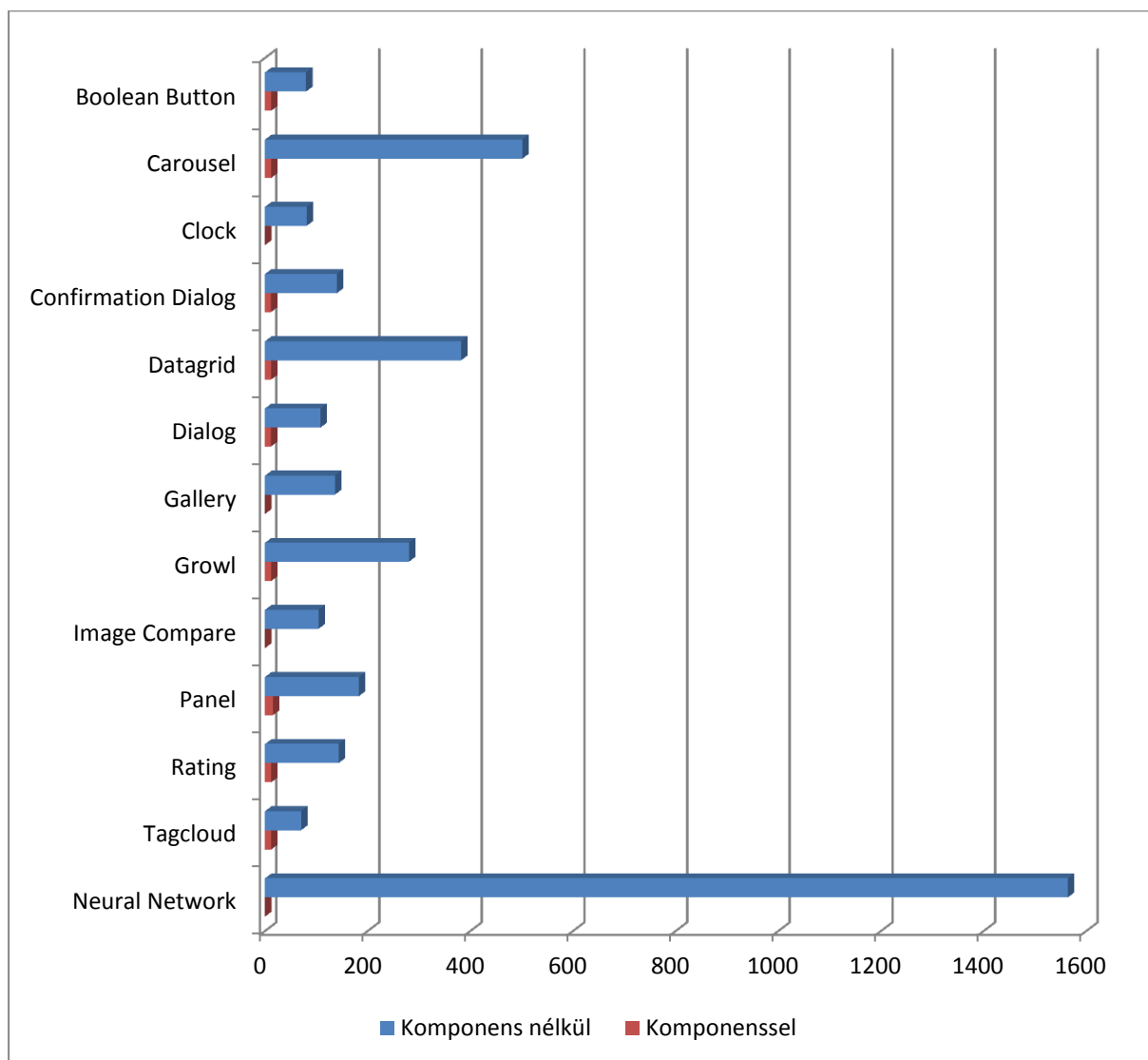
A dialog komponensek esetén a dialog tartalmában lévő gombok eseménykezelői növelik a Dart kód sorainak számát a második oszlopban. Az adatok a confirmation dialog komponens esetén 2 gomb (pl. Yes és No) eseménykezelőjét, a sima dialognál pedig egy gomb (pl. OK) eseménykezelőjét tartalmazzák.

Természetesen a jobb oldali oszlop adatai még lejjebb is vihetők, ha például az eseménykezelőkhöz a rövidebb, lambdás alakot használok.

A 25. ábra vizuálisan jeleníti meg az előző táblázat adatait.

A 3. táblázatban a HTML kódot vizsgáltam meg. Az első két oszlop a HTML fa mélységét tartalmazza komponensek nélkül és komponensekkel, az utolsó két oszlop pedig hasonlóan a HTML kód sorainak hosszát tartalmazza. A mélység 0, ha csak egyetlen elem szükséges a komponens használatához.

Ezekből az adatokból is egyértelműen látszik, hogy a komponensek egyszerűsítik a HTML kódot, hiszen nélkülük hosszabb és mélyebb HTML-re lenne szükség, komponensekkel viszont csak a paraméterezéshez szükséges néhány attribútumot és gyerek elemet kell leírni, a többi HTML részlet rejtve marad a felhasználótól.



25. ábra: A szükséges Dart kód sorainak száma

A sorok számát úgy kell érteni, hogy minden attribútum egy sorban szerepel az elemével – ez nem túl jól olvasható kódot eredményez, de egységessé teszi az adatok rögzítését.

A komponens nélküli adatokhoz a template elemen belüli tartalmat vettem alapul, hiszen ezt kellene használni, ha nem lenne minden egyetlen komponensbe zárva.

A komponenseket feltételező számadatok tartalmazzák az item template-et azoknál a komponenseknél, amelyeknél definiálható item template (pl. carousel), de az egyszerűség kedvéért csak egyetlen elemet tartalmazó item template-ekkel számoltam.

Ezen kívül ha egy komponens lehetőséget biztosít egyéb elemek definiálására (pl. header és footer a panel esetében), akkor ezeket is egy-egy elemmel definiáltam.

Azoknál a komponenseknél, ahol tetszőleges gyerek elem megadható (pl. gallery, tagcloud), az adatok rögzítéséhez 3 elemmel számoltam.

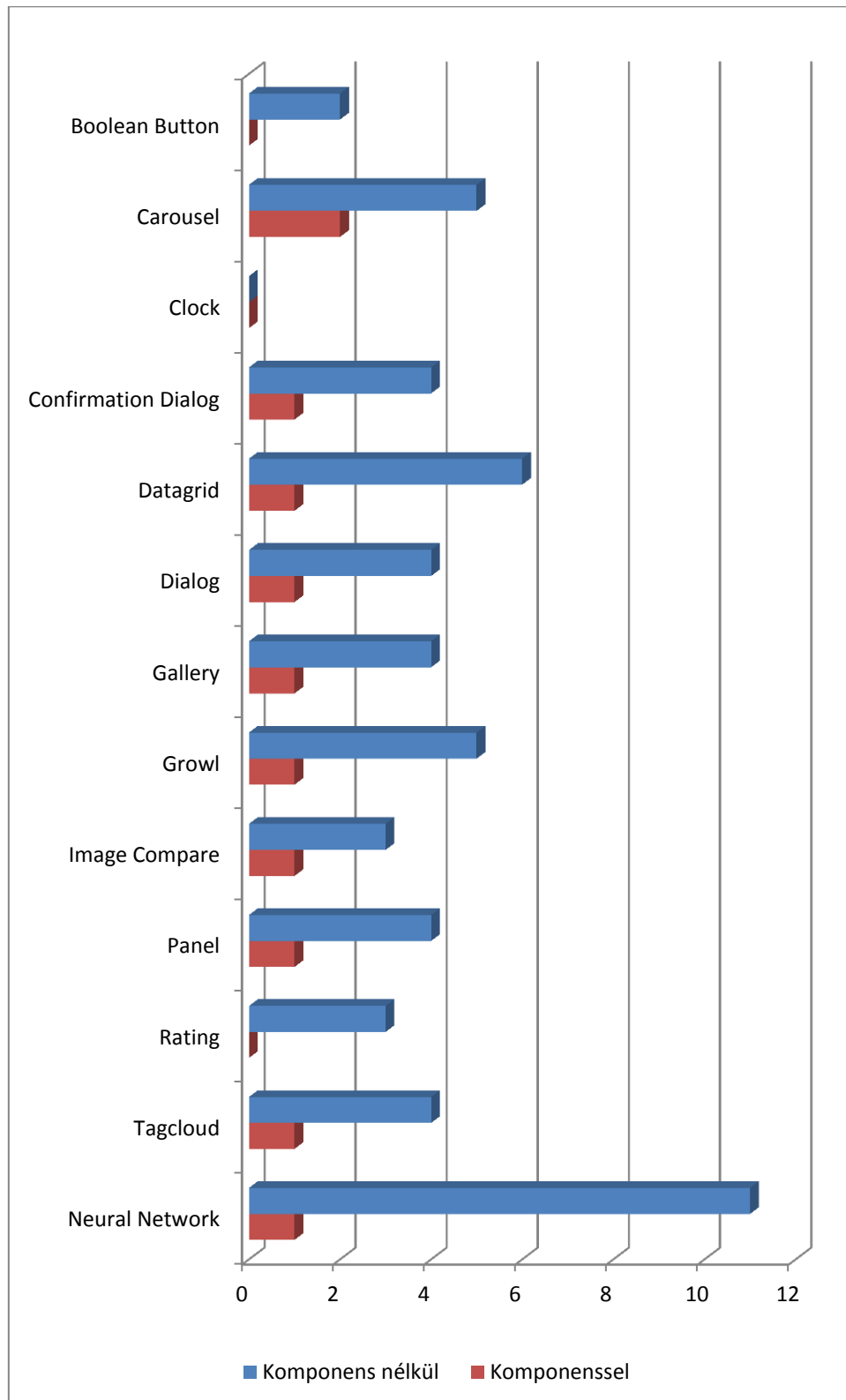
	HTML FA MÉLYSÉGE		SOROK SZÁMA	
	Komponens nélkül	Komponenssel	Komponens nélkül	Komponenssel
Boolean Button	2	0	6	1
Carousel	5	2	33	7
Clock	0	0	1	1
Confirmation Dialog	4	1	18	6
Datagrid	6	1	21	3
Dialog	4	1	14	4
Gallery	4	1	13	5
Growl	5	1	16	3
Image Compare	3	1	15	4
Panel	4	1	21	5
Rating	3	0	10	1
Tagcloud	4	1	9	5
Neural Network	11	1	184	8

3. táblázat: A szükséges HTML maximális mélysége és sorainak száma

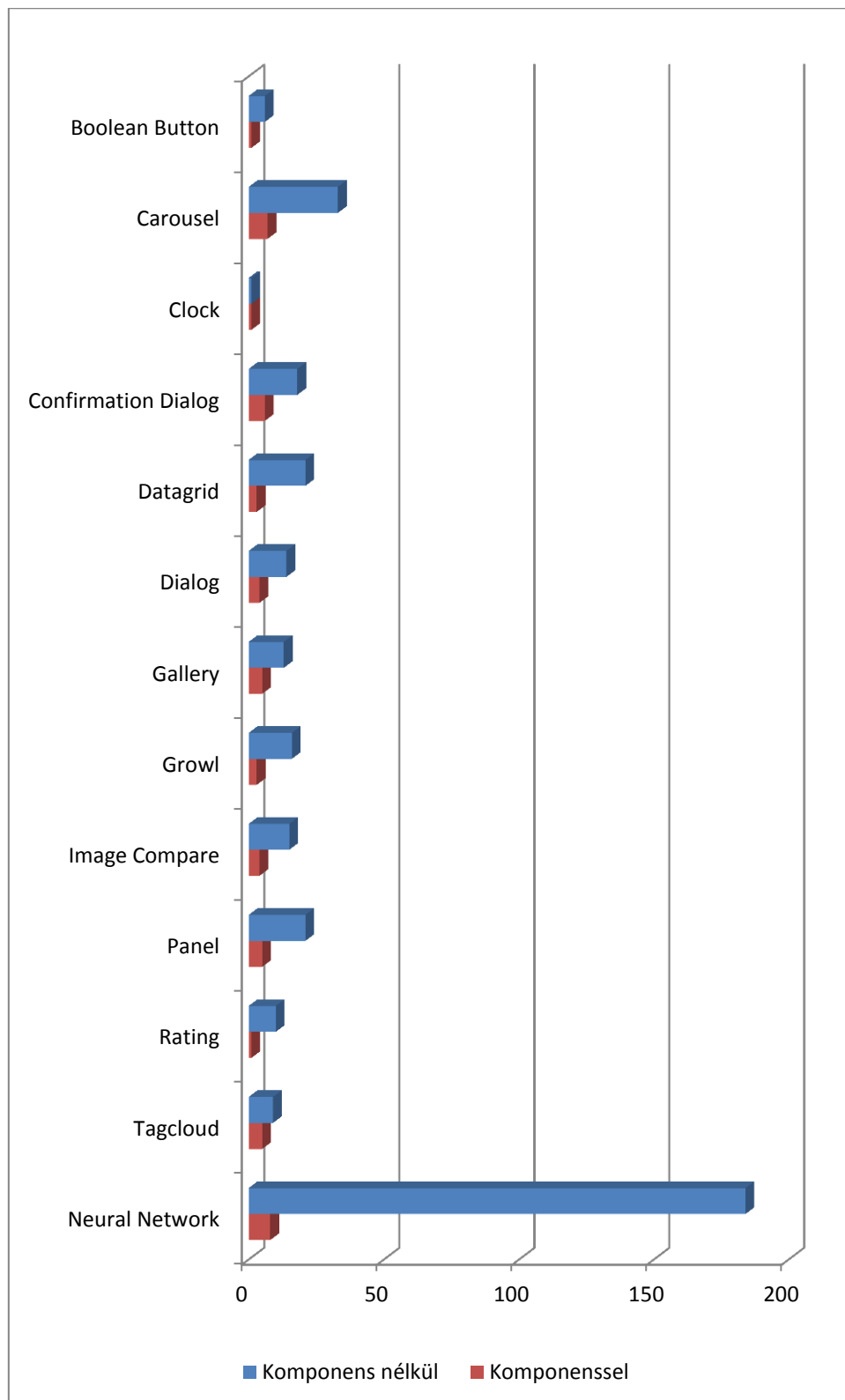
Fontos azonban megjegyezni, hogy bár a komponenssel minden esetben rövidebb HTML kód is elégséges, a sorok száma, illetve a HTML mélysége tartalmazza a model-driven view szabvány által definiált adatkötési lehetőségeket. Ez azt jelenti, hogy a valójában megjelenített DOM bővebb is lehet, mint az itt mért logikai DOM.

A sorok számába természetesen beleszámít a CSS sorainak száma is, de ennek mérése nem lenne egzakt, hiszen a megjelenítés finomságától és bonyolultságától függ, hogy mennyire komplex stíluslapok szükségesek.

A 26. ábra a HTML kód mélységének diagramját, míg a 27. ábra a szükséges sorok számának alakulását mutatja.



26. ábra: A szükséges HTML kód mélysége



27. ábra: A szükséges HTML kód sorainak száma

A 4. táblázat a fenti értékek alapján számított compression ratio metrikákat tartalmazza százalékban megadva. Látható, hogy a komponensekkel elért kódtömörítési arány jelentősnek mondható, a Dart kód sorainak számára átlag 5,89%, a HTML sorainak számára 30,13%, míg a HTML mélységére 18,78% jön ki.

	DART	HTML	
	Sorok száma	Sorok száma	Mélység
Boolean Button	15	16,67	0
Carousel	2,4	21,21	40
Clock	0	100	0
Confirmation Dialog	7,86	33,33	25
Datagrid	3,14	14,29	16,67
Dialog	10,19	28,57	25
Gallery	0	38,46	25
Growl	4,29	18,75	20
Image Compare	0	26,67	33,33
Panel	8,2	23,81	25
Rating	8,39	10	0
Tagcloud	17,14	55,56	25
Neural Network	0	4,35	9,09

4. táblázat: A web komponensek compression ratio értékei százalékban mérve

5.3. Teljesítménynövekedés

A szükséges sorok számának drasztikus csökkenése természetesen nem jelenti azt, hogy a különbség nem jelenik meg például hálózati forgalomban, hiszen a szükséges funkcionális a komponensek letöltésekor ugyanúgy megjelenik a böngészőben, csak a komponens használója számára ez transzparens.

Azt is megvizsgáltam, hogy a betöltési sebesség hogyan viszonyul az általam két legjobbnak tartott jelenleg is használható komponens keretrendszerhez képest. Ez a két keretrendszer a kliens oldali Ext JS, illetve a szerver oldali JSF PrimeFaces.

Az 5. táblázat tartalmazza a betöltési időket milliszekundumban mérve a két keretrendszer, illetve a Dart web komponensek esetén. A web komponenseket Google Chrome (3. oszlop) és Dartium (4. oszlop) böngészőkben teszteltem – előbbiben a dart2js fordító által gene-

rált Javascript betöltési idejét láthatjuk, míg az utolsó oszlop a Dart kód Dartiumba való betöltésének sebességét tartalmazza.⁵

A táblázat adataiból kitűnik, hogy mivel az Ext JS egy nehézsúlyú keretrendszer, sok scriptet kell betöltenie, mire a komponensei használhatóak lesznek, ezért viszonylag sok időbe kerül a betöltődése a PrimeFaceshez képest.

A dart2js által generált Javascript kód lassabban töltődik be, mint a PrimeFaces, ez a technológiák korai stádiuma miatt várható is volt, hiszen a polyfilek betöltődése is időbe telik. Meglepetés azonban, hogy a gallery és growl komponensek esetén fordított a helyzet.

A carousel és datagrid komponensek elég bonyolultak ahhoz, hogy a generált Javascript kód hosszabb és komplexebb legyen, mint a többi esetben.

	Ext JS	PrimeFaces		
Boolean Button	261	182	207	149
Carousel	359	193	1540	241
Clock	–	–	168	154
Confirmation Dialog	342	199	260	178
Datagrid	390	204	1590	257
Dialog	335	212	272	166
Gallery	–	302	235	199
Growl	308	286	282	178
Image Compare	–	199	260	178
Panel	285	205	243	155
Rating	243	176	210	148
Tagcloud	276	180	260	164
Neural Network	–	–	476	347

5. táblázat: A komponensek betöltési idejének összehasonlítása az Ext JS és PrimeFaces keretrendszerekkel

A Dartiumban lévő Dart VM képes a Dart kódot is futtatni, ami nyilvánvalóan tömörebb, és gyorsabban betöltődik, így azt állapíthatjuk meg, hogy a Dartium esetében a betöltési idő rövidebb, mint a Chrome-ban, több esetben még a PrimeFacesnél is gyorsabb.

Ezek az eredmények biztatóak, viszont nem kell itt megállnunk: a HTML5 application cache [43] lehetővé teszi, hogy egy manifest fájlban megjelöljük, hogy mely fájlokat kell a

⁵ Néhány komponensnek nincs megfelelője az egyes keretrendszerekben, így ezekhez az esetekhez nem tartozik adat.

böngészőnek eltárolnia. Mivel a komponensek viszonylag statikusak, megtehetjük, hogy a hozzájuk szükséges fájlokat cache-eltetjük, így tovább közelíthetjük a beépített elemek betöltési sebességét.

A 6. táblázat a cache használata mellett mutatja a betöltési sebességeket a Chrome és Dartium böngészők esetén.

		
Boolean Button	179	102
Carousel	1430	158
Clock	152	98
Confirmation Dialog	228	115
Datagrid	1470	177
Dialog	235	112
Gallery	218	156
Growl	254	115
Image Compare	204	98
Panel	217	99
Rating	181	101
Tagcloud	232	109
Neural Network	408	267

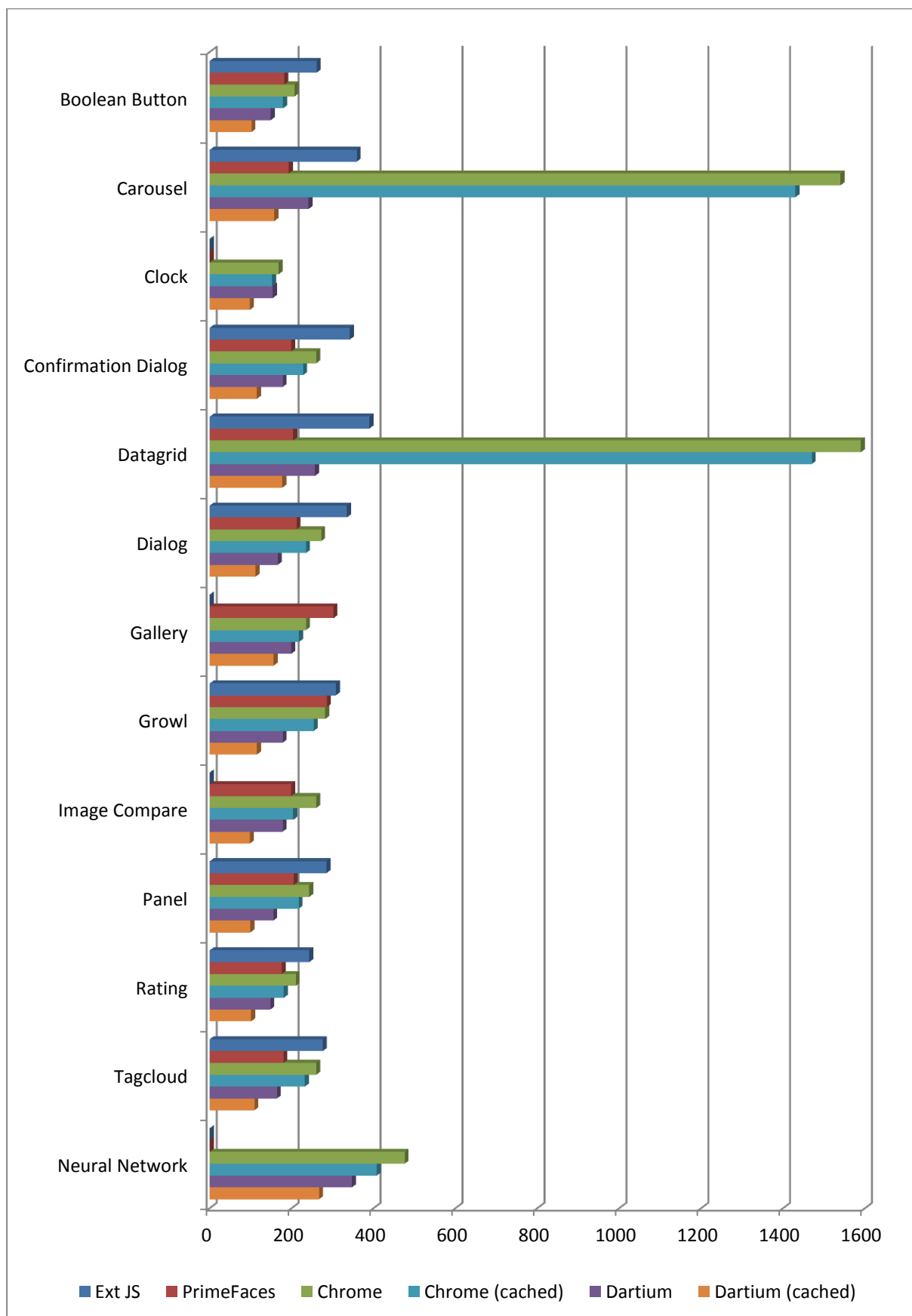
**6. táblázat: A komponensek betöltési ideje
HTML5 application cache használatával**

Megállapíthatjuk, hogy a cache-elt változatok minden esetben gyorsabban töltődnek be, mint a megfelelő párjuk – ez várható is volt, hiszen megspóroljuk a hálózati forgalmat.

A cache bevezetésével a Chrome sebessége megközelíti, több esetben meg is haladja a PrimeFaces keretrendszerét, ami nagyon biztató jel, tekintettel arra, hogy ezekben az adatokban a polyfillék betöltési ideje is szerepel.

Néhány esetben azonban – például a carousel és datagrid komponenseknél – valószínűleg a cache méretkorlátja miatt a Javascript fájl nem cache-elhető, így csak kicsivel csökkenthető a betöltési idő. Ez a dart2js fordító által generált Javascript kód optimalizálásával javítható.

A fenti táblázatok adatait a 28. ábra grafikonja szemlélteti.



28. ábra: A web komponensek cache-elt és cache nélküli betöltési ideje

6. Összefoglalás

Az 5. fejezet alapján egyértelműen állíthatjuk, hogy a web komponensek nemcsak használhatóság, hanem teljesítmény szempontjából is biztató eredményeket mutatnak.

Nem arról van szó, hogy a jelenlegi technológiák használhatatlanok lennének, hiszen akkor nem születtek volna kiváló HTML alapú webes alkalmazások. A probléma abból adódik, hogy a fejlesztők kénytelenek hozzászokni, hogy kerülőutakat használjanak, és Javascriptben valósítsák meg igényeiket, ahelyett, hogy a böngészőt „tanítanánk meg” az új követelményekre.

Az új szabványok ebben segítenek, hiszen a böngésző kibővítésével a futtató környezet tudni fog a template-ekről és web komponensekről, így arra is lehetőség nyílik, hogy optimalizáltan kezelje őket. Nyilvánvalóan azok a kliens oldali keretrendszerek, amelyek jelenleg biztosítanak hasonló funkciókat, nem tudnak alacsony szinten dolgozni, így a teljesítmény kárt szenved.

Ezt nemcsak a Google ismerte fel: a HTML Templates szabvány egyik alkotója (Tony Ross) például a Microsoft munkatársa, ezen kívül a Mozillának is vannak hasonló törekvései (X-Tags, [44]). Ahogyan azt már a HTML evolúciója esetében megszokhattuk, ebben az esetben is valószínűleg a végleges szabvány a nagy böngészőgyártók legjobb ötleteit fogja ötvözni.

A szabványokat természetesen már ma is használhatjuk polyfilllek segítségével. Bár ezek lassabb működést tesznek lehetővé, mintha a böngészők natívan támogatnák a web komponensek funkcionalitását, az 5.3. alfejezet adatai bizonyítják, hogy a sebességbeli különbség nem számottevő, és egy kis odafigyeléssel (HTML5 application cache) tovább csökkenthető.

Amint a natív támogatás megjelenik, az új alkalmazások egyre bátrabban használhatják majd az új feature-öket, és biztos vagyok benne, hogy más régebbi, komplex alkalmazások (pl. Gmail) is át fogják ezeket venni.

Ennek következménye pedig, hogy idővel maga a web is egyre szemantikusabb lesz, hiszen a formázáshoz szükséges div és span elemek helyett jelentéssel bíró web komponenseket láthatunk majd az oldalakon.

Az új technológiák tehát egyértelműen a webes alkalmazásfejlesztők előnyére fognak válni, és valószínűleg a meglévő keretrendszerek is át fognak venni belőlük részeket. Ezen felül azonban az új fejlesztési módszerek megjelenése is előrelépés lesz az alkalmazásfejlesztésben.

A komponensek használata már más nyelvekben is elterjedt, hiszen hamar felismerték, hogy az osztályoknál és objektumoknál szükség van magasabb szintű építőkövekre is. Megje-

lentek az olyan tervezési minták, mint az MVC és különböző változatai, és ezek beépültek a már meglévő technológiákba.

A kidolgozott elvek még egy lépést tesznek a könnyebb karbantarthatóság és tesztelhetőség irányába, hiszen segítségükkel az MVC komponenseket is különböző kategóriákba sorolhatjuk: az elsődleges komponensek felhasználhatnak másodlagos és model komponenseket is annak érdekében, hogy komplex adatokat és funkcionalitást fogadjanak be a külvilágtól – ezzel egyszerűen megvalósulhat az adat- és kódintegráció.

A komponensek HTML nyelvbe való bevezetésével valószínűleg meg fognak jelenni általánosan használható komponens könyvtárak, amelyek fejlesztői alacsony szintű kiterjesztési pontokat adhatnak a komponensek felhasználói számára, kiegészítve a könyvtárakat alapértelmezett funkcionalitást megvalósító komponensekkel. Így az esetek többségében nem lesz szükség saját komponensek írására, a fejlesztőknek elég lesz csak beimportálni és felhasználni a szükséges elemeket.

A web komponensek által képviselt előnyök méréséhez kidolgozásra került egy új compression ratio szoftvermetrika, melynek bevezetését az indokolja, hogy a hagyományos metrikák ilyen magas szintű technológiák esetén vagy nem használhatóak, vagy jelentős módosításra szorulnak. A tömörítési arány viszont bármely technológiai szinten alkalmazható, hiszen nincs másra szükségünk, csak egy programra és egy metrikára, melyek segítségével kiszámíthatjuk, hogy a metrika hogyan javul, ha az új technológiát alkalmazzuk.

Jelen esetben meghatározhatjuk, hogy a web komponensek mennyiben javítják például a szükséges sorok számát, a HTML kód hosszát és mélységét, vagy egyéb olyan metrikát, amely mérhető az alacsonyabb szintű és az új technológia esetén is.

Fontos azonban megemlíteni, hogy a dolgozatban bemutatott technológiák még változhatnak a végleges szabványok elfogadásáig, így a kódrészletek végül módosításra szorulhatnak. A technológiákkal ellentétben azonban az alapelvek és tervezési módszerek a jövőben is megállják majd a helyüket, így azokat a végleges szabványok esetén is alkalmazhatjuk majd.

Az elért eredmények tehát a következőkben foglalhatók össze:

- Új fejlesztési modell kidolgozása, mely segítségével HTML nyelven használhatunk fel és integrálhatunk magas szintű komponenseket.
- Új szoftvermetrika bevezetése a különböző technológiai szintek összehasonlításához.
- Komponens könyvtár létrehozása a web komponensek előnyének demonstrálásához és a szoftvermetrikák méréséhez.
- Egy web komponens alapú demonstrációs alkalmazás létrehozása mesterséges neurális hálók belső működésének vizualizálásához és elemzéséhez.

Melléklet

A következő kódrészletek azt mutatják, hogyan lehet egy egyszerű web komponenst elkészíteni Dart nyelven.

A `click_counter.html` fájl tartalmazza a komponens belső DOM szerkezetét. A komponens egy gombból, egy span elemből és egy felületre kötött attribútumból áll. A gomb megnyomására lefut a komponens `increment()` metódusa, a span elemet pedig a komponens létrehozásakor gyerek elemként kell megadni.

A komponens hivatkozik arra a Dart fájlra (`click_counter.dart`), amely a funkcionalitását definiálja.

```
<!DOCTYPE html>
<html>
  <body>
    <element name="click-counter" constructor="ClickCounter" extends="div">
      <template>
        <button on-click="increment()">Click me</button>
        <content select="span"></content>
        {{count}}
      </template>

      <script type="application/dart" src="click_counter.dart"></script>
    </element>
  </body>
</html>
```

A komponenst leíró osztály 0-ra inicializálja a `counter` attribútum értékét, és létrehoz egy `countChanged` eseményt, amelyet a gombra való kattintáskor bocsátunk ki, a `counter` attribútum új értékével. Amikor a komponenst a böngésző elhelyezi a DOM-ban, a konzolra kiírjuk a „*Component inserted!*” szöveget.

```

import "package:web_ui/web_ui.dart";
import "dart:html";
import "dart:async";

class ClickCounter extends WebComponent {
  int count = 0;

  static const EventStreamProvider<CustomEvent> _COUNT_CHANGED_EVENT =
    const EventStreamProvider<CustomEvent>("countChanged");
  Stream<CustomEvent> get onCountChanged => _COUNT_CHANGED_EVENT.forTarget(this);
  static void _dispatchCountChangedEvent(Element element, int count) {
    element.dispatchEvent(new CustomEvent("countChanged", detail: count));
  }

  void inserted() {
    print("Component inserted!");
  }

  void increment() {
    _dispatchCountChangedEvent(this, ++count);
  }
}

```

A komponenst felhasználó oldal importálja a komponens HTML fájlját, és elhelyez egy példányt a DOM-ban, a `count` attribútum értékét 3-ra inicializálva, és megadva a szükséges `span` elemet.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Click Counter</title>
    <link rel="import" href="click_counter.html"/>
  </head>
  <body>
    <div is="click-counter"
      count="{{3}}">
      <span>Current count:</span>
    </div>

    <script type="application/dart" src="index.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </body>
</html>

```

Az oldal dart fájlja a main függvényben feliratkozik a komponens eseményére, és minden alkalommal, amikor azt a komponens kibocsátja, kiírja az új értéket a konzolra.

```

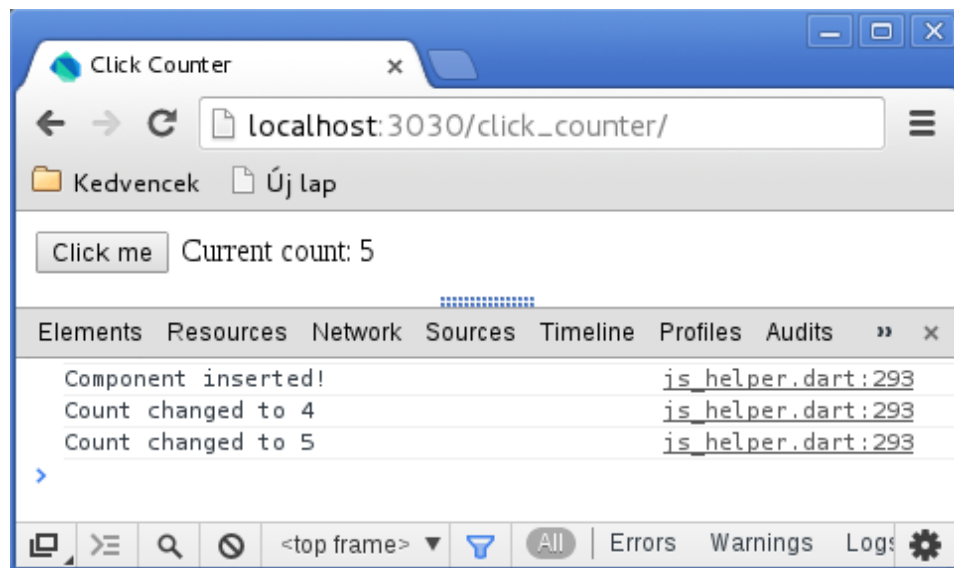
import "dart:html";
import "dart:async";
import "click_counter.dart";

void main() {
  Timer.run(() {
    DivElement clickCounterDiv = query('div[is="click-counter"]');
    ClickCounter clickCounter = clickCounterDiv.xtag;

    clickCounter.onCountChanged.listen((CustomEvent event) {
      print("Count changed to ${event.detail}");
    });
  });
}

```

A következő ábra az oldal működését szemlélteti. Jól látszik a komponens belső szerkezete, az attribútum aktuális értéke két kattintás után, illetve az inicializáló üzenet és az eseménykezelők üzenete a konzolban.



Irodalomjegyzék

- [1] „Wikipedia: HTML,” 2013. [Online]. Available: <http://en.wikipedia.org/wiki/HTML>.
- [2] W3C HTML Working Group, „XHTML 1.0 The Extensible HyperText Markup Language (Second Edition),” 2002. [Online]. Available: <http://www.w3.org/TR/xhtml1/>.
- [3] W3C HTML Working Group, „HTML 5.1 Nightly,” 2013. [Online]. Available: <http://www.w3.org/html/wg/drafts/html/master/>.
- [4] Microsoft, „XAML in WPF,” [Online]. Available: <http://msdn.microsoft.com/en-us/library/ms747122.aspx>.
- [5] Oracle, „JavaFX 2 Documentation,” 2011. [Online]. Available: <http://docs.oracle.com/javafx/>.
- [6] „Overview of the Qt QML Module,” 2013. [Online]. Available: <http://qt-project.org/doc/qt-5.0/qtqml/qtqml-index.html>.
- [7] Oracle, „JavaServer Faces Technology,” [Online]. Available: <http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>.
- [8] ICEsoft Technologies Inc., „ICEFaces,” 2013. [Online]. Available: <http://www.icesoft.org/java/projects/ICEfaces/overview.jsf>.
- [9] Apache Software Foundation, „Apache MyFaces Project,” 2013. [Online]. Available: <http://myfaces.apache.org/>.
- [10] Oracle, „Oracle Application Development Framework - Oracle ADF,” 2013. [Online]. Available: <http://www.oracle.com/technetwork/developer-tools/adf/overview/index.html>.
- [11] PrimeTek, „PrimeFaces - Ultimate JSF Component Suite,” 2011. [Online]. Available: <http://primefaces.org/>.
- [12] Microsoft, „ASP.NET Web Forms,” 2013. [Online]. Available: <http://www.asp.net/web-forms>.
- [13] Google, „AngularJS - HTML enhanced for web apps!,” [Online]. Available: <http://angularjs.org/>.
- [14] „Knockout,” [Online]. Available: <http://knockoutjs.com/>.
- [15] The jQuery Foundation, „jQuery UI,” 2013. [Online]. Available: <http://jqueryui.com/>.
- [16] Sencha, „Ext JS,” 2013. [Online]. Available: <http://www.sencha.com/products/extjs>.
- [17] G. Hempton, „The Top 10 Javascript MVC Frameworks Reviewed,” [Online]. Available: <http://codebrief.com/2012/01/the-top-10-javascript-mvc-frameworks-reviewed/>.
- [18] R. Sharp, „What is a Polyfill?,” 2010. [Online]. Available: <http://remysharp.com/2010/10/08/what-is-a-polyfill/>.
- [19] D. Glazkov, „Shadow DOM,” 2013. [Online]. Available: <http://www.w3.org/TR/shadow-dom/>.
- [20] D. Glazkov, „What the Heck is Shadow DOM?,” 2011. [Online]. Available: <http://glazkov.com/2011/01/14/what-the-heck-is-shadow-dom/>.

- [21] D. Glazkov, R. Weinstein és T. Ross, „HTML Templates,” 2013. [Online]. Available: <http://www.w3.org/TR/html-templates/>.
- [22] S. Ladd, „Your First Model Driven View with Dart,” [Online]. Available: <http://blog.sethladd.com/2012/11/your-first-model-driven-view-with-dart.html>.
- [23] D. Glazkov, „Custom Elements,” 2013. [Online]. Available: <http://www.w3.org/TR/custom-elements/>.
- [24] „CoffeeScript,” 2013. [Online]. Available: <http://coffeescript.org/>.
- [25] Google, „Dart,” [Online]. Available: <http://www.dartlang.org/>.
- [26] Google, „The Dart Programming Language Specification,” 2013. [Online]. Available: <https://www.dartlang.org/docs/spec/latest/dart-language-specification.html>.
- [27] Google, „Dart Performance,” 2013. [Online]. Available: <https://www.dartlang.org/performance/>.
- [28] J. Heaton, Introduction to Neural Networks for Java, 2nd Edition, St. Louis: Heaton Research, Inc., 2008.
- [29] C. Gershenson, „Artificial Neural Networks for Beginners,” 2003. [Online]. Available: <http://arxiv.org/ftp/cs/papers/0308/0308031.pdf>.
- [30] A. Venkataraman, „The Backpropagation Algorithm,” 1999. [Online]. Available: <http://www.speech.sri.com/people/anand/771/html/node37.html>.
- [31] S. R. Chidamber és C. F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, Vol. 20, 1994.
- [32] T. J. McCabe, Complexity Measure, IEEE Transactions on Software Engineering, Volume 2, 1976.
- [33] „Halstead Metrics,” IBM, [Online]. Available: <http://pic.dhe.ibm.com/infocenter/rtrthelp/v8r0m0/index.jsp?topic=%2Fcom.ibm.rational.testtrt.studio.doc%2Ftopics%2Fcsmlhalstead.htm>.
- [34] L. Marco, „Measuring software complexity,” *Enterprise Systems Journal* , 1997.
- [35] „Wikipedia: Cohesion (Computer Science),” 2013. [Online]. Available: [http://en.wikipedia.org/wiki/Cohesion_\(computer_science\)](http://en.wikipedia.org/wiki/Cohesion_(computer_science)).
- [36] „Wikipedia: Source Lines of Code,” 2013. [Online]. Available: http://en.wikipedia.org/wiki/Source_lines_of_code.
- [37] T. Klemola és J. Rilling, „A Cognitive complexity metric based on Category learning,” *Proceedings of EEEE*, 2003.
- [38] N. Tak, „Generation of a New Complexity Dimension Scheme for Complexity Measure of Procedural Program,” *International Journal of Computer Science and Information Technologies*, 2013.
- [39] F. Tsui és O. Karam, „Essentials of Software Engineering,” 2011.
- [40] S. Sencer, „Complexity Metrics and Models,” [Online]. Available: <http://yunus.hun.edu.tr/~sencer/complexity.html>.
- [41] L. Fortnow, „Kolmogorov Complexity,” 2010.
- [42] J. Kun, „Kolmogorov Complexity - A Primer,” 2012. [Online]. Available: <http://jeremykun.com/2012/04/21/kolmogorov-complexity-a-primer/>.
- [43] E. Bidelman, „A Beginner's Guide to Using the Application Cache,” 2010. [Online]. Available: <http://www.html5rocks.com/en/tutorials/appcache/beginner/>.

- [44] Mozilla, „X-Tag - The Custom Elements Polylib,” [Online]. Available: <http://www.x-tags.org/>.
- [45] M. H. Halstead, „Elements of Software Science,” 1977.