Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Polymorph per realizzare estensioni di browser multi-piattaforma

Relatore:
Chiar.mo
Fabio Vitali

Presentata da:
Mirko Ferranti

Sessione III 2019-2020

Indice

Capitolo 2
Introduzione

Internet è una delle invenzioni tecnologiche che rivoluzionò maggiormente la società in questi ultimi decenni, tale da rendere il suo utilizzo, fin dagli anni 90, estremamente utile per le persone che ne usufruiscono grazie ai browser, applicazioni pensate per facilitare la navigazione e visualizzazione dei siti web; non a caso infatti il primo browser di successo, Mosaic, ebbe un tale successo da interessare le aziende a questa nuova tecnologia. In particolare, una caratteristica ormai imprescindibile per queste applicazioni è la possibilità di integrare le estensioni, installate dagli utenti sia per motivi estetici che organizzativi in modo da soddisfare maggiormente i loro desideri.

L'evoluzione delle estensioni fu estremamente rapida: da programmi slegati dall'esperienza di navigazione, pensati più che altro per aggiungere funzionalità aggiuntive tramite scorciatoie, permisero ben presto, grazie all'adozione di nuovi standard e architetture, non solo una maggiore simbiosi tra esse e il browser, ma anche un loro forte legame con l'esperienza di navigazione provata dagli utenti tant'è che attualmente possono alterare l'aspetto dei siti visitati in base alle loro impostazioni.

Proprio perché l’evoluzione delle estensioni fu particolarmente densa di standard tecnici differenti, la programmazione di un estensione è fortemente legata all’architettura del browser interessato dalla pubblicazione di tale programma; nonostante in questi ultimi tempi ci sia la tendenza ad adottare un modello di estensioni indipendente dalla struttura del browser, questa transizione è ancora ancorata a una fase iniziale, mentre buona parte dei browser più utilizzati supportano le estensioni di Chrome grazie al codice sorgente Chromium. Tuttavia ciò rende comunque le estensioni disponibili per Chrome fortemente legate all’architettura di Chromium.

Polymorph, l’applicazione coinvolta in questa tesi, è un'applicazione pensata per agevolare, da parte del programmatore, la transizione dal modello iniziale su cui è basata l’estensione interessata adattandola a un nuovo modello pensato per un browser differente. L’utilizzo di Polymorph può essere di grande aiuto per il programmatore poiché la documentazione delle estensioni, a causa della loro rapida evoluzione, può diventare obsoleta nell’arco di poco tempo, specialmente se non vengono consultati i siti ufficiali dedicati alla loro programmazione.

Polymorph, nel supportare il programmatore, non solo adatta l’estensione interessata a un nuovo modello, ma offre anche, tramite il file di istruzioni, i passaggi principali coi quali debuggare la nuova applicazione sul nuovo browser, utile se il programmatore è estraneo alle sue modalità di sviluppo o se sta iniziando l’approccio verso il mondo della programmazione delle estensioni. Oltre a ciò, Polymorph, tramite la nuova estensione, può offrire un'idea più completa dell’architettura di un estensione nel caso essa sia basata su uno standard poco documentato.

Per spiegare tali differenze nel mercato dei browser è necessario esporre, prima di tutto, la loro storia che, seppur sia molto recente avendo le sue radici nel WorldWideWeb di Tim Berners-Lee del 1989, vanta un evoluzione molto veloce come i siti web fruibili tramite queste applicazioni: la possibilità di inserire link ipertestuali coi quali organizzare un sito o riportare le fonti da cui si attingono le informazioni e la possibilità di essere informati velocemente sul mondo circostante, rendendolo ancora più connesso di quanto potesse essere possibile tramite i mass media fino a quel momento disponibili.

Difatti i browser, in particolar modo Mosaic che fu il primo a godere di un ottimo successo, furono delle killer application tali da modificare notevolmente la società degli anni ‘90: gli utenti furono stupiti dalla reperibilità di qualsiasi informazione tramite pochi click e di conseguenza anche le aziende dimostrarono notevole interesse alle potenzialità di queste tecnologie portando a una crescita esponenziale di siti e delle loro visite da parte degli utenti.

L’evoluzione del mercato spiccò in particolare quando la concorrenza ebbe come attori principali Netscape Navigator, colui che riuscì a conquistare il pubblico portando all’abbandono di Mosaic, e Internet Explorer nella seconda metà degli anni ‘90: se da una parte Netscape riuscì a mantenere per un certo periodo di tempo il predominio, d’altra parte Internet Explorer procurò velocemente molta utenza sia per la distribuzione del software in forte legame col sistema operativo Windows che per l’introduzione di funzionalità innovative come il supporto al CSS consentendo un’ottima personalizzazione grafica dei siti.

Dopo un lungo periodo dove Internet Explorer fu il browser più utilizzato, il mercato mutò grazie a nuovi protagonisti del mercato: Mozilla Firefox, Safari e Google Chrome, molto apprezzati dall’utenza grazie a una maggiore attenzione verso la sicurezza e le novità proposte per la navigazione. Allo stato attuale Google è l’attore più importante non solo per il browser proprietario Chrome , ma anche per il progetto open-source Chromium da cui deriveranno altri browser rilevanti.

Conclusa l’analisi storica, la tesi espone le caratteristiche principali dei browser al momento più in voga dipendente da due elementi: il loro scopo, l’obiettivo posto dallo sviluppatore per trasmettere un'idea riguardo l’esperienza di navigazione, e l’implementazione di nuove funzionalità. Queste ultime sono spesso gli strumenti col quale lo sviluppatore potrà trasmettere la sua idea di esperienza di navigazione, tant’è che ogni browser può avere una filosofia estremamente differente dalla concorrenza puntando alla sicurezza, all’ottimizzazione delle prestazioni o l’originalità.

Vengono introdotti anche i concetti generali delle estensioni, le loro funzionalità più importanti e interessanti enunciando pure i possibili rischi riguardo la sicurezza e un'esposizione storica sulla loro nascita e evoluzione. Questa esposizione sul mercato attuale dei browser e sulla loro filosofia verrà approfondita nel capitolo 2.

Completata la panoramica sul mercato e la storia dei browser, la tesi si focalizza su Chrome e Safari, i due browser maggiormente coinvolti da Polymorph. Si inizia con una visione d'insieme tale da scorgere le loro differenze nello scopo, nei loro traguardi conseguiti e anche il legame che Chrome ebbe per un pò di tempo con Safari grazie all'utilizzo di WebKit, per poi distaccarsi dalla sua filosofia adottandone una propria per restare al passo con gli ultimi ritrovati della programmazione web.

Le due case di sviluppo perseguono linee di pensiero differenti riguardo lo sviluppo dei browser e il loro punto di vista riguardo la disponibilità open source delle loro applicazioni: Chrome, grazie alla vasta comunità di sviluppatori, tende a ricevere moltissime idee che verranno integrate nei frequenti aggiornamenti destinati agli utenti; Apple, invece, rilascia poche major release per Safari, destinando gran parte degli aggiornamenti su bugfix che correggono problemi di sicurezza.

Nessuno dei due browser preso in esame ha il codice totalmente open source, entrambi possiedono una porzione piuttosto importante di codice proprietario. Chrome offre un progetto totalmente open source denominato Chromium; il suo codice sorgente, dall'interfaccia sino agli elementi più tecnici, può essere riutilizzato. Apple, d'altro canto, offre la possibilità di utilizzare solo il framework WebKit, comprendente il motore HTML e il motore Javascript.

Riguardo i motori HTML e Javascript, entrambi i browser furono molto importanti contribuendo allo sviluppo di siti web sempre più elaborati a livello grafico e esecutivo. Chrome si basa su Blink, fork del motore HTML di WebKit pensata per seguire una filosofia differente di esecuzione multiprocesso, e V8, il primo motore Javascript basato sulla compilazione Just-In-Time consentendo l'ottimizzazione delle performance sui siti fortemente basati su questo linguaggio.

Il framework proposto da Safari, WebKit, include JavaScriptCore e WebCore: quest'ultimo in particolare, assieme ai motori HTML di Firefox e Opera, diede un forte impulso all'adesione dei protocolli proposti da W3C. WebCore riuscì a passare in poco tempo i test Acid2 e Acid3, raccogliendo molti consensi. La tesi si preoccupa anche di esporre il funzionamento dei motori HTML e Javascript.

Infine verranno esposti i due modelli di estensioni proposti dai due attori: le Chrome Extension e le Safari App Extension, definendo la loro storia e le loro differenze di progettazione, piuttosto sostanziali anche nel loro scopo essendo state pensate per programmatori con conoscenze differenti. L'analisi di Chrome e Safari sarà ulteriormente approfondita nel capitolo 3.

Successivamente la tesi tratta l'argomento principale, le estensioni di Chrome e Safari, enunciando in particolar modo la difficoltà, considerando un'estensione aderente a uno dei due standard, nel passare a uno standard differente rispetto a quello originario. È qua che viene introdotto Polymorph, introducendo il suo funzionamento tramite l'interazione con la sua interfaccia e perché sia stato sviluppato in un determinato modo.

La differenza implementativa data dai due modelli di estensioni è dovuta dalla struttura differente dei loro progetti, dalle conoscenze differenti riguardanti i linguaggi da utilizzare e anche dalle procedure differenti di debugging. L'unico elemento che hanno in comune sono le politiche per loro progettazione: sia Google che Apple consigliano caldamente una politica single-purpose, con la quale le estensioni compiono esclusivamente una funzionalità ben specifica.

Le estensioni di Chrome, di cui questa tesi parlerà nello specifico della versione Manifest V2 essendo quella considerata da Polymorph, non sono estensioni con una struttura rigida dei file: al di là delle interfacce e script integrati nel progetto, l'unico file obbligatorio è il manifest.json che permette l'identificazione dell'estensione, configurare le autorizzazioni e scegliere quali script far intervenire sui siti visitati durante la navigazione.

Tuttavia ciò non esclude una buona strutturazione per la progettazione di un estensione di Chrome: suddividere i file in cartelle con nome esplicativo, al loro tipo e in base allo scopo a cui sono predisposti aiuta a comprendere più facilmente il contributo di essi ai fini delle funzionalità offerte. Inoltre questa buona pratica di progettazione aiuta anche il debugging e il miglioramento dell'estensione.

Le Safari App Extension, d'altro canto, sono strutturate rigorosamente e al loro interno non contengono solo l'estensione vera e propria, ma anche un eseguibile predisposto alla configurazione dell'estensione all'interno di Safari. Pacchettizzare

queste due applicazioni in un unico progetto permette al browser una migliore identificazione dell'estensione installata grazie al certificato offerto dall'eseguibile.

Vengono esposti i problemi principali riguardanti la conversione da un modello a un altro: le difficoltà non consistono solo nell'utilizzo di linguaggi di programmazione differenti, ma anche per la differente rigorosità su certe meccaniche di programmazione come il casting di tipo e riguardo la simbiosi tra il browser e l'estensione, implementata in modo differente a seconda dello standard considerato. Per ulteriori informazioni su come questi passaggi siano non banali e perchè Polymorph possa offrire contributi importanti il capitolo 4 offrirà maggiori dettagli al riguardo.

La tesi ha intenzione di dimostrare che tramite Polymorph il processo di conversione di un estensione preesistente venga agevolato grazie alla sostituzione di istruzioni incompatibili con uno certo standard in favore di istruzioni compatibili. Questa agevolazione è sia a livello di programmazione che di progettazione, grazie al file di istruzioni il quale fornisce i passaggi opportuni. La tesi si focalizza maggiormente sulle agevolazioni riguardanti la programmazione, quelle riguardanti la sostituzione di istruzioni.

L'estensione utilizzata per il testing di Polymorph è Joo Janta 200, un'estensione pensata per sostituire delle stringhe presenti nei siti durante la navigazione con stringhe alternative. Sia le stringhe da cercare che quelle sostitutive sono configurabili liberamente dall'utente manualmente o caricando una tabella proveniente da Google Sheets. Joo Janta 200 sfrutta le principali caratteristiche presenti nei modelli odierni di estensioni: utilizzo di una memoria locale, modifiche dei siti tramite script esterni e forte simbiosi tra estensione e browser.

L'estensione originale sfrutta molte istruzioni native offerte da Chrome e sfrutta i vantaggi di alcune autorizzazioni presenti nel suo manifest.json. Le autorizzazioni, in buona parte dei casi, sono facilmente convertibili in analoghe adatte per le Safari App Extension, tuttavia il caricamento dell'interfaccia HTML richiede l'utilizzo di istruzioni in linguaggio Swift. La conversione tiene conto dei campi strettamente necessari per l'esecuzione, quindi gli elementi descrittivi sono tralasciati.

La memoria locale di Chrome è uno strumento estremamente utile dato che, tramite autorizzazioni opportune, rende disponibile tale memoria anche allo script esterno; Joo Janta 200 fa pieno utilizzo di questa funzionalità. Questa funzionalità non è disponibile per le Safari App Extension, rendendo necessario un meccanismo di scambio dei messaggi rigoroso a causa della separazione più netta tra l'estensione e il browser. Polymorph saprà convertire correttamente questo passaggio implementativo.

Polymorph effettua correttamente anche la conversione al contrario, da una Safari App Extension a una estensione di Chrome, in questo caso sostituendo lo scambio dei messaggi col meccanismo della memoria locale presentato dal Joo Janta 200 originale. Tuttavia la conversione non è perfetta poiché la lettura sintattica delle istruzioni, da una parte, non è così potente e dall'altra Polymorph non può risolvere gli errori logici di esecuzione. Il capitolo 5 presenterà nel dettaglio le istruzioni e autorizzazioni più importanti da convertire.

Infine la tesi espone un sunto riguardo il futuro di Polymorph, riassumendo i vantaggi principali e le problematiche che verranno corrette in un prossimo futuro. Lo sviluppo dell'applicazione è interessato anche ai nuovi standard recentemente in auge riguardo lo sviluppo di estensioni per i browser.

Capitolo 3
Similarità e differenze tra browser

3.1 In questo capitolo si espone un excursus sui principali browser attualmente presenti sul mercato, denotando la loro evoluzione e come si siano guadagnati un certo successo grazie alle funzionalità inedite che introdussero rispetto alla concorrenza.

3.1 Storia dei browser

In questo paragrafo tratteremo la storia dei browser in ordine temporale, partendo dalle sue origini fino ai giorni nostri.

Le prime idee per la realizzazione di un browser risalgono al 1990, quando Tim Berners-Lee, all'interno del sistema operativo NeXTSTEP programmò l'applicazione WorldWideWeb (o Nexus) a scopo dimostrativo [1]; il motivo per cui venne scelto tale sistema operativo consiste in particolare per le librerie integrate che facilitano la programmazione di applicazioni grafiche. Nel medesimo periodo vennero pensati anche i protocolli principali di quello che diverrà il World Wide Web, il linguaggio di markup HTML e il protocollo HTTP.

Nexus venne pensato in particolare modo per la condivisione facilitata di documenti riguardanti ricerche scientifiche, essendo stato sviluppato per il CERN di Ginevra e, oltre a contenere un editor di file HTML, potè già vantare molte funzionalità che diverranno basilari per i browser attuali tra l'apertura di link ipertestuali, il caricamento di immagini, una versione primordiale dei segnalibri e i bottoni "avanti" e "indietro" per cambiare più velocemente la visualizzazione di un documento già visitato, introducendo un concetto primordiale della cronologia . Nel 1991 Tim realizzò Line Mode Browser, un browser eseguibile tramite prompt che, rinunciando ad alcune funzionalità di Nexus, permise l'accesso ai siti anche con macchine con sistemi differenti da NeXTSTEP [2].

Il 1993 fu un anno molto importante per la diffusione di Internet e per la programmazione dei browser: infatti, oltre a essere stato pubblicato il codice sorgente di Nexus in modo che le sue funzionalità possano essere ereditate [3], arrivò il primo browser che ottenne buon successo anche al di fuori degli ambiti accademici, Mosaic [4].

I motivi del successo di Mosaic furono molti: in primis l'installazione semplificata rispetto ai browser precedenti, il rendering delle immagini all'interno del documento anziché in una finestra esterna, il supporto a molti protocolli di navigazione e l'implementazione di un'interfaccia grafica intuitiva, seppur non sia il primo browser "grafico" essendo stato preceduto da Erwise [5]; le funzionalità di Mosaic e la sua semplicità di utilizzo permisero un utilizzo sempre più massiccio della rete nell'uso quotidiano e l'interesse economico da parte delle aziende riguardo questa nuova tecnologia [6].

A partire dal 1994 un browser che ebbe un'ottima accoglienza fu Netscape Navigator che, seppur non abbia nulla in comune con Mosaic a livello di codice, eredita le sue funzionalità migliorandole aggiungendo novità tuttora alla base del web come l'animazione delle GIF e l'introduzione di Javascript per rendere dinamici i siti web, fino a quel momento totalmente statici [7].

Le novità di Netscape Navigator lo resero presto il browser più utilizzato grazie alla sua pubblicazione sui sistemi operativi più diffusi in quel momento, generando anche controversie legate all'implementazione di standard proprietari o per le violazione della privacy legate ai cookie, altra funzionalità introdotta da Netscape

spesso utilizzata dai siti di commercio [8]. Nel 1995 nacque anche Opera, originariamente MultiTorg Opera, browser pensato in particolare per coloro che possedevano un sistema operativo differente da Windows.

Tra i vari browser che vennero rilasciati durante il dominio di Netscape uno particolarmente importante fu Internet Explorer, il browser sviluppato dalla Microsoft: nonostante le due prime versioni fossero assai inferiori rispetto a Netscape e furono strettamente basate sul codice sorgente di Spyglass Mosaic [9], la terza versione, rilasciata nel 1996 e notevolmente differente dalle precedenti, fu il primo browser a introdurre il supporto ai file CSS [10], ai controlli ActiveX e contenuti multimediali, rendendo i siti molto più elaborati nel contenuto e nell'aspetto [11].

Oltre a ciò introdusse PICS, protocollo primitivo mirato per identificare i contenuti adulti presenti nei siti; l'insieme di queste novità rese Internet Explorer 3 il primo browser a occupare una importante fetta di mercato fino a quel momento ricoperta da Netscape dando inizio, nella seconda metà degli anni '90, a una concorrenza molto aggressiva contesa tra quest'ultima e Microsoft riguardo i browser e gli standard che essi introdussero: il periodo che ricoprì la concorrenza agguerrita delle due software house viene spesso definito "prima guerra dei browser", distinguendola da un altro periodo che ebbe inizio nel 2004 con concorrenti differenti, anch'esso definito da una concorrenza aggressiva, definito "seconda guerra dei browser".

Tuttavia, nonostante Internet Explorer portasse con sè novità interessanti, Netscape continuò ad essere l'attore più rilevante sul mercato [12]. Il mercato dei browser si fece molto più accesa nel 1997 col rilascio, da parte delle due aziende di sviluppo, di Netscape Communicator, suite di applicazioni comprendente Netscape Navigator 4, un programma per la gestione della posta elettronica e un editor HTML, e Internet Explorer 4, il nuovo browser proposto da Microsoft.

Da una parte la nuova versione di Netscape Navigator ebbe molti crash e problemi di rendering delle pagine contenenti CSS [13] dato che il browser supportò un formato concorrente di minore successo, Javascript Style Sheet o JSSS [14], dall'altra Internet Explorer continuò ad allargare notevolmente il suo bacino d'utenza sia per le tantissime nuove funzionalità che vanno dalle favicon, a un

sistema più avanzato di parental control fino a un nuovo motore di rendering, Trident [15].

Tra i motivi principali del successo di Explorer fu anche la controversa politica di distribuzione del software tramite un aggiornamento del sistema operativo Windows: se precedentemente Explorer era un'applicazione autonoma, dal rilascio della quarta versione Internet Explorer sarà fortemente integrato con Windows (e obbligatorio a partire da Windows 98) impossibilitando la disinstallazione del browser agli utenti che volessero utilizzare un software alternativo.

La distribuzione di Internet Explorer 5 nel 1999 portò una vittoria ancora più netta da parte di Microsoft nei confronti di Netscape, ciò è anche dovuto a causa di Netscape 6, software inizialmente molto instabile, causando un danno d'immagine non indifferente alla software house poichè basato sul codice sorgente di Mozilla Application Suite non ancora pronto, in quel momento, per essere distribuito a un pubblico generalista [16].

Mozilla Application Suite proviene da Mozilla Organization, organizzazione no profit istituita nel 1998 e finanziata da Netscape con lo scopo di rilasciare e rendere open source il codice sorgente dei suoi browser in modo da poter essere supervisionato e migliorato; tra le novità che risaltano in particolare fu la progettazione del motore HTML Gecko [17].

Tra le novità che portarono al successo Internet Explorer 5 furono il rendering notevolmente ottimizzato delle pagine e l'introduzione di XMLHTTPRequest che diede inizio all'idea dello stile di programmazione AJAX permettendo un comportamento dei siti web estremamente dinamico [18], tuttavia anche questa versione portò con sé polemiche non solo per la politica di distribuzione, ma anche, come capitò con Netscape, per l'introduzione di molti standard proprietari, portando molti siti a essere visualizzati correttamente solo tramite Internet Explorer 5, costringendo la stessa Microsoft, tramite le versioni successive del browser, a introdurre quirks mode coi quali visualizzare in maniera corretta i siti estremamente dipendenti dalle funzionalità offerte da IE 5 [19].

Di conseguenza la prima metà degli anni '00 vide un monopolio praticamente totale di Internet Explorer, principalmente a causa della enorme diffusione di Windows e, di conseguenza, del browser fortemente integrato col sistema

operativo: i principali concorrenti, Netscape e Opera, si affidavano a un bacino d'utenza estremamente ridotto e fecero causa a Microsoft a causa della politica di distribuzione di Internet Explorer, considerata illegale [20]; tra l'altro quest'ultima accusò Microsoft di bloccare l'accesso agli utenti del loro browser a servizi come MSN.com [21].

Inoltre Internet Explorer 6, fin da quando fu distribuito nel 2001, fu aspramente criticato per vari motivi:

- Il browser ebbe parecchi problemi a livello di sicurezza e di stabilità: anche righe molto semplici di HTML poterono provocare un crash dell'applicazione.
- Di conseguenza, a causa del monopolio, Microsoft non si preoccupò di aggiornare tempestivamente l'applicazione nonostante le numerose segnalazioni dei bug.
- Sempre a causa della mancanza di una concorrenza aggressiva, Microsoft non si preoccupò ad aggiornare il browser ai nuovi protocolli di programmazione web come le nuove versioni di CSS o un supporto completo della trasparenza delle immagini PNG.

Questi problemi diedero inizio a numerose campagne informando gli utenti dei rischi riguardanti l'utilizzo di Internet Explorer consigliando caldamente l'installazione di un browser differente [22].

Nella prima metà degli anni '00 Mozilla continuò per lungo tempo lo sviluppo di Mozilla Application Suite, successivamente separata in varie applicazioni in base alle loro differenti funzionalità. Tra esse figura Mozilla Firefox, inizialmente chiamata Phoenix, che riscosse un enorme successo quando venne resa disponibile, nel 2004, la versione 1.0 per vari motivi:

- Il blocco dei popup integrato e la preoccupazione dell'applicazione riguardo la navigazione sicura furono acclamati, fino al punto di promuoverlo come sostituto di Internet Explorer.
- Mozilla Firefox fu molto più attento all'adesione dei nuovi protocolli di navigazione proposti dal W3C, evitando la creazione di standard proprietari.

- A differenza del resto della concorrenza, Firefox fu distribuito gratuitamente senza l'ausilio di pubblicità.

Questi motivi, uniti ai lenti rilasci di patch per Internet Explorer, permisero a Firefox di guadagnare velocemente molti utenti (nel 2007 verrà usato dal 25% delle persone) [23].

Pure Safari, browser della Apple rilasciato nel 2003, ebbe un buon successo per la velocità e l'attenzione riguardo l'adesione ai nuovi standard per la programmazione web rendendo di enorme successo il motore WebKit, tuttavia Safari ebbe diffusione più limitata rispetto a Firefox principalmente per la sua distribuzione, essendo disponibile solo nelle macchine con sistema operativo Mac OS X (per un breve periodo venne distribuito anche per Windows, seppur aspramente criticato per le numerose falle di sicurezza)[24].

Safari è inoltre uno dei primi browser di successo di cui venne fatto il porting sugli smartphone, estendendo le funzionalità che questi ultimi avevano rispetto ai telefoni cellulari permettendo anche la sincronizzazione con altri dispositivi Apple dello stesso utente i suoi dati di navigazione come i segnalibri.

Nel 2008 subentrò un altro attore importante sul mercato dei browser, Google, tramite Google Chrome; dopo un periodo iniziale con il motore WebKit, nel 2013 il team di sviluppo decise di fare un fork del motore, denominandolo Blink; Blink, grazie all'architettura riprogettata, permise modifiche sostanziali del suo codice sviluppare più facilmente nuove funzionalità.

Uno dei motivi principali che decretarono il successo di Chrome è la frequenza degli aggiornamenti, assai più elevata rispetto ai browser concorrenti, aumentando non solo le sue funzionalità, ma anche la prevenzione di bug o implementazione di misure di sicurezza. Chrome, assieme a Safari, fu uno dei primi, tramite WebKit, a supportare HTML5 che, tramite nuovi tag, rese possibile l'implementazione di nuove funzionalità precedentemente implementabili solo tramite i plug-in, in particolar modo i contenuti multimediali.

Inoltre la velocità del browser nell'esecuzione di script Javascript elaborati come quelli di Google Maps favorì ulteriormente il successo dell'applicazione, ravvivando il mercato. La rinnovata concorrenza aggressiva, prima proveniente da

Safari e Firefox e successivamente anche da Chrome, portò l'utenza ad abbandonare Internet Explorer in favore dei concorrenti, con funzionalità ammodernate, prestazioni ottimizzate e una maggiore attenzione per la sicurezza [25].

Al momento il predominio del mercato figura a Google non solo grazie a Chrome, usato da circa il 60% dell'utenza nel marzo 2020 [26], ma anche grazie al progetto open-source Chromium, sul quale è basata la maggioranza dei browser sia tra quelli totalmente nuovi che per quelli già presenti come Opera [27], che abbandonarono il loro modello proprietario in favore delle funzionalità di Chromium in parte per la velocità dei suoi aggiornamenti e in parte per il supporto al suo vasto catalogo di estensioni.

Per citare un ulteriore esempio del successo di Chromium, Microsoft Edge, rilasciato nel 2015 basato su un motore proprietario derivato da Internet Explorer denominato EdgeHTML, abbandonò questa soluzione in favore di Chromium nel 2019 [28], tant'è che, tra i browser più utilizzati attualmente, solo Safari e Firefox possiedono un'architettura e motori differenti da quelli di Google.

Tra i nuovi browser che stanno emergendo, particolare risonanza stanno ricevendo Brave e Vivaldi, entrambi browser basati su Chromium, Blink e V8 (seppur Vivaldi abbia parte del codice programmato tramite Node.js per alcune funzionalità): Brave è pensato in particolare per la tutela della privacy e della sicurezza integrando anche una politica innovativa per il supporto dei content creator [29], mentre Vivaldi ha lo scopo di essere estremamente personalizzabile secondo i desideri dell'utente offrendo un'esperienza di navigazione differente da quella offerta dalla concorrenza [30].

3.2 Browser attualmente più diffusi

Allo stato attuale la concorrenza dei browser vede vari protagonisti che si sono guadagnati col tempo molti utenti. In questo capitolo si presenteranno i browser dal meno utilizzato al più utilizzato tramite le statistiche di marzo 2021 stimate da W3Counter [31]. Considerando tutte le versioni di Internet Explorer e escludendo

Edge, sempre in base a tali statistiche, esso copre 1,8% del mercato [32], leggermente maggiore di Opera, per cui si partirà da quest'ultimo.

Opera

Opera è il più vecchio browser aggiornato con regolarità e vanta molte particolarità rispetto a ciò che offrirono e offrono i concorrenti. In primis Opera, rilasciato nell'aprile del 1995, è il browser che mantenne più a lungo una politica di vendita a pagamento del software, mantenendola per ben 10 anni fino al 2005, periodo dove ormai tutti i browser furono gratuiti [33]. Riferendosi sempre alle statistiche di W3Counter, Opera ebbe il picco del suo successo nel 2017, dove ad agosto ebbe il 5% del mercato [34].

Opera è un pioniere nell'inserimento di nuove funzionalità oggi giorno indispensabili per i browser, in primis la possibilità di visitare più pagine web in una sola finestra tramite la visualizzazione in tab, il primo a poter salvare le sessioni di navigazione e uno dei primi a offrire una enorme personalizzazione della sua interfaccia tramite estensioni e temi.

Attualmente rimane un browser molto apprezzato per funzionalità comode come il raggruppamento di tab in cartelle o per un debugger estremamente sofisticato, tale da modificare in tempo reale il codice HTML o visualizzarlo sotto l'ottica di un browser puramente testuale. Opera, inoltre, integra anche altre funzionalità come la posta elettronica o sfruttare la modalità turbo, avvantaggiando il caricamento delle pagine tramite connessioni lente.

Opera ebbe enorme successo anche nell'ambito dei browser dedicati a dispositivi mobile con schermo ridotto grazie a funzionalità che consentono la compressione automatica di contenuti troppo lunghi per gli schermi all'interno di pulsanti, in modo da migliorare la leggibilità delle pagine tramite questi dispositivi. Opera ebbe particolare successo col motore di rendering Presto, introdotto nel 2003, uno dei primi a supportare lo standard DOM e regole CSS avanzate e uno dei motori di rendering più veloci.

Dal 2013 il browser è basato su Blink, il motore di rendering HTML di Chrome: questo da un lato rimosse molte funzionalità uniche del browser e rese le sue estensioni incompatibili con l'applicazione, ma aiutò Opera a supportare le

estensioni di Chrome e si vide anche una maggiore stabilità del motore di rendering, uno dei problemi principali che afflisse Presto.

Questa rimozione importante di funzionalità portò alla nascita di Vivaldi che, nonostante anch'esso abbia come motori Blink e V8, venne pensato per riprendere le funzionalità innovative che Opera introdusse grazie a Presto, consentendo una enorme personalizzazione del browser agli utenti concedendo un esperienza di navigazione notevolmente differente da quella offerta dalla concorrenza.

Internet Explorer

Internet Explorer è un altro browser, rilasciato nell'agosto del 1995, che ebbe un'importante influenza sul mercato dei browser, in primis per l'integrazione totalmente gratuita del software all'interno del sistema operativo Windows; da una parte ciò portò sia a un notevole successo del browser surclassando Netscape Navigator e i concorrenti portando per un lungo periodo Internet Explorer a un monopolio tale da conquistare il 95% del mercato nel 2004 osservando le statisitche raccolte da TheCounter.com [35], monopolio che verrà intaccato solo a partire dal 2005, dall'altra grandissima parte dei siti web all'epoca progettati vennero programmati in modo da ottimizzare la loro visualizzazione tramite le caratteristiche offerte da Internet Explorer. Tuttavia il software suscitò anche svariate polemiche che variano dalla politica di commercializzazione, alla sua fragilità riguardo la sicurezza e la mancanza di aderenza agli standard richiesti dal W3C: citando un esempio storico, Internet Explorer impiegò molto tempo a implementare le nuove funzionalità di CSS e standard W3C tali da visualizzare correttamente la pagina di test Acid2. Tuttavia, nonostante ciò e l'abbandono del software in favore di Microsoft Edge, Internet Explorer è tuttora tra i browser più utilizzati, in parte dagli utenti che possiedono versioni di Windows differenti da Windows 10 che, in particolare, dalle aziende che progettarono siti estremamente dipendenti dalle funzionalità di Internet Explorer, specialmente da ActiveX, dove la loro riprogettazione può risultare molto costosa in termini di tempo e costi [36].

Microsoft Edge

Microsoft Edge è pensato come un browser totalmente differente da Internet Explorer, più attento alle politiche di sicurezza e all'adesione sugli standard più recenti proposti dal W3C senza inventare propri protocolli come successe col

browser precedente di Microsoft. Inizialmente, al suo rilascio avvenuto nel 2015, l'architettura venne basata su un fork dei motori di Internet Explorer adattandoli alle ultime novità riguardo la navigazione e col motore EdgeHTML pensato per essere integrato facilmente nelle applicazioni Windows allo scopo di visualizzare pagine HTML comprese di allegati. Tuttavia, dal 2019, le versioni più recenti di Edge si basano su un browser totalmente ricostruito in modo da integrare Blink e V8, i motori HTML e Javascript di Chrome, permettendo agli utenti di usufruire delle estensioni di Chrome edi sue altre funzionalità [37]. Il periodo di maggiore successo del browser, secondo W3Counter, è giugno 2020 con il 4,8% del mercato.

Mozilla Firefox

Mozilla Firefox, rilasciato nel 2004 con la versione 1.0, è considerato l'erede spirituale di Netscape Navigator (da cui il nome Phoenix delle sue prime versioni) e fin dal suo rilascio venne acclamato per la sua filosofia open-source che gli permise da una parte di aderire velocemente agli standard del W3C, dall'altra a inserire funzionalità molto richieste e apprezzate dagli utenti come il blocco dei popup o la possibilità di estendere le funzionalità del browser tramite estensioni molto sofisticate. Oltre a queste funzionalità, il browser vanta un debugger avanzato. Il suo motore di rendering è Gecko, motore utilizzato per la prima volta nel 1998 tramite la sesta versione di Netscape Navigator; dal 2017 il motore HTML è migliorato tramite un progetto supervisionato da Mozilla denominato Quantum, pensato per adeguare Gecko ai nuovi standard come HTML5, ridefinire l'interfaccia del browser [38]. Grazie al progetto Quantum, Gecko è ottimizzato per le CPU multicore mentre il motore Javascript SpiderMonkey fu uno dei primi a implementare un compilatore JIT per ottimizzare le performance. Firefox ebbe molto successo nel 2010, occupando il 34,1% del mercato a luglio [39].

Safari

Safari, il browser progettato dalla Apple rilasciato nel 2003, è stato pensato per offrire un browser di default ai loro dispositivi che potesse sfruttare al massimo le loro prestazioni grazie all'utilizzo di istruzioni native; ma Safari è un importante pioniere pure riguardo l'aderenza agli standard W3C, spesso risultando il primo browser ad aderire appieno ai loro nuovi protocolli, e nella sua filosofia open-source, essendo i suoi motori HTML e Javascript fortemente derivati da

KHTML e KJS della comunità KDE, spesso condividendo le loro novità con quest'ultima apportando notevoli progressi all'esperienza di navigazione. Una delle principali caratteristiche di Safari rispetto ai concorrenti, oltre alle migliori prestazioni nei Mac e Iphone, è il forte legame tra il browser e altri servizi offerti da Apple come iCloud, col quale si possono sincronizzare i segnalibri, i tab aperti o altre funzionalità riguardanti la sincronizzazione della sessiona attualmente aperta con gli altri dispositivi. Il periodo più fortunato per Safari fu il 2016, quando a gennaio riuscì a conquistare il 20,7% del mercato [40].

Google Chrome

Google Chrome, rilasciato nel 2008, è il principale attore del mercato attuale dei browser a causa delle innovative funzionalità offerte, in particolare è il primo browser che credette nella programmazione elaborata di script Javascript grazie a un compilatore JIT che ottimizzò notevolmente l'esecuzione di script scritti con quel linguaggio; ciò portò velocemente anche gli altri browser ad adottare motori Javascript dall'architettura simile e di conseguenza siti web assai più basati su un Javascript ormai perfettamente in grado di sostituire i plug-in come Flash, che da quel momento in poi vennero abbandonati. Un altra caratteristica inedita di Chrome sono gli aggiornamenti più regolari grazie a un programma di sviluppo rigido e l'inserimento di tante funzionalità inedite tra cui un nuovo modello col quale progettare le estensioni di ottimo successo tale da essere supportato dai browser concorrenti, abbandonando i loro modelli di progettazione delle estensioni. Attualmente è il browser più utilizzato grazie anche a un'interfaccia pulita e al tempo stesso ampiamente personalizzabile sia a livello grafico che a livello di navigazione grazie alle numerose estensioni disponibili. Nonostante sia già da lungo tempo il browser di maggiore successo, i risultati più ragguardevoli risalgono al 2019, occupando il 65,4% del mercato a marzo [41].

3.3 Browser extension

Da molto tempo gli utenti che usano abitualmente i browser non usano questi software solo ed esclusivamente per visitare pagine, ma desiderano anche organizzare in maniera personalizzata il software in modo che possa soddisfare le loro esigenze. Le estensioni, software allegabili al browser solitamente ben visibili nella sua interfaccia grazie alla presenza di un bottone con una sua icona o un

toolbar, adempiono a questo obiettivo grazie all'implementazione di funzionalità che i browser non possiedono nativamente oppure in maniera limitata.

Alcune delle estensioni, addirittura, furono talmente influenti nelle funzionalità offerte tali da rendere queste ultime uno standard per i browser oggi presenti [42]. Citando un paio di esempi, molto influente fu il blocco delle pubblicità offerto da Adblock Plus, estensione per Mozilla Firefox pubblicata nel 2005 migliorando le funzionalità di un altra estensione Firefox, Adblock, da non confondere con l'omonima estensione pubblicata per Google Chrome nel 2009 [43]. Adblock Plus è una delle estensioni di maggiore successo tanto che, nel 2008, l'estensione per Mozilla Firefox venne usata quotidianamente da almeno tre milioni di utenti [44]. Attualmente molti browser, come Chrome dal 2018 [45], implementano un sistema di blocco delle pubblicità, con efficacia variabile a seconda dell'applicazione considerata.

Oppure, altra estensione che ebbe una notevole importanza, fu Google Toolbar rilasciato nel 2000 per Internet Explorer 5, pensato per offrire all'utente i risultati del loro motore di ricerca [46]. Oggi giorno i browser interpretano automaticamente le stringhe e, in base alla loro composizione, decidere se accedere direttamente al sito identificato da esse o riportarle al motore di ricerca predefinito. Le possibilità offerte dalle estensioni, man mano sempre più ampie grazie all'implementazione di nuovi standard, al giorno d'oggi possono differire tantissimo e le più diffuse riguardano la privacy.

Le estensioni possono agire su molti elementi nella navigazione dell'utente; di seguito è presente un elenco delle principali funzionalità delle estensioni per browser:

- Privacy e sicurezza: molte estensioni vengono sviluppate per migliorare la sicurezza mentre si visualizzano le pagine web (per esempio, bloccando le finestre pop-up).

- Funzionalità aggiuntive che si possono visualizzare con un click; tra di esse rientrano estensioni che informano l'utente sulle previsioni meteo, comparatori dei prezzi di un prodotto venduto su più siti di e-commerce, traduttori, calendari, gestione dei download e tante altri elementi che possono interessare l'utente.

- Blocco delle pubblicità: molto diffuse sono le estensioni che bloccano le pubblicità, seppur siano molto criticate dagli editori dei siti web essendo le pubblicità una delle principali fonti di guadagno, tant'è che sempre più siti adottano controlli "anti-blocco" pregando l'utente di disabilitare queste estensioni.

Le estensioni sono diverse dai plugin: infatti se entrambi raggiungono il medesimo scopo (ovvero, aggiungere funzionalità ai browser e alla navigazione), differiscono nella modalità; le estensioni sono infatti sempre disponibili all'utente e sono in codice sorgente, mentre i plugin sono eseguibili che eseguono contenuti programmati tramite un software presenti nei siti web, per esempio Adobe Flash Player permette la visualizzazione di contenuti programmati in Adobe Flash.

Se tempo fa entrambi ebbero una buona diffusione, oggi giorno, per motivi di sicurezza, i browser supportano sempre meno i plugin favorendo le estensioni [47]. Nonostante ciò pure queste ultime possono essere vettori di malware oppure raccogliere senza il consenso dell'utente i suoi dati sensibili [48], infatti le estensioni hanno notevoli libertà che possono inficiare la navigazione dell'utente: inserimento di pubblicità, modifiche indesiderate alle impostazioni del browser o richieste HTTP nascoste.

Le estensioni, in particolare se installate da siti di terze parti, possono essere dei malware creati con lo scopo di rubare i dati dell'utente o installare adware; un'estensione può addirittura modificare le pagine web e i reindirizzamenti ai login per bypassare l'autenticazione a due fattori implementata sui siti web [49]. La storia delle estensioni che tratteremo sarà strutturata in ordine temporale, partendo dalle sue origini fino agli standard più recenti.

Le estensioni per i browser furono introdotte nel 1999 da Internet Explorer 4, ma le loro funzionalità erano limitate solitamente alla personalizzazione dell'interfaccia aggiungendo barre di ricerca e altre funzionalità limitate non legate alla navigazione nella rete [50]. Infatti fu Firefox il primo browser a concedere ampia libertà alle estensioni: infatti, le sue estensioni, allora chiamate add-on e facenti uso di interfaccia XUL e script Javascript, potevano implementare funzionalità più elaborate e complesse come il debugging del codice HTML o maggiore personalizzazione dell'interfaccia alterando il suo aspetto grafico.

Le estensioni di Firefox furono il primo modello ad ottenere un enorme successo, tale da raggiungere i tre miliardi di download nel 2012 [51]. Attualmente, Firefox ha abbandonato il modello degli add-on supportando quello delle estensioni di Chrome, tuttavia consiglia caldamente, per la programmazione di estensioni compatibili col browser, di seguire il modello proposto dal W3C, le WebExtension API, a livello di architettura simili alle estensioni di Chrome, ma con nuove istruzioni che possano fare le veci di quelle native spesso fondamentali per le Chrome Extension in modo da poter rendere queste applicazioni fruibili anche per i browser non basati su Chromium [52].

Chrome introdusse nel 2009 un nuovo modello di estensioni che rivoluziona la loro programmazione tale che buona parte dei browser, in tempi successivi, aggiunse il supporto di esse; infatti, l'utilizzo dei linguaggi HTML, CSS e Javascript, i linguaggi più utilizzati nella programmazione web, uniti a una documentazione dettagliata, rese le estensioni molto più facili da progettare, non richiedendo lo studio di linguaggi ad hoc per la loro implementazione; di conseguenza tale modello rese facile la disponibilità di molti tipi di estensioni nell'arco di poco tempo [53].

Dal 2015 gran parte della concorrenza, Firefox e i browser che decisero di adottare Chromium come Edge, supporta le estensioni di Chrome, quindi eventuali incompatibilità possono verificarsi solo se il browser non si aggiorna alle loro novità implementative. Le estensioni per Chrome sono diventate man mano sempre più complesse sia nello scambio di messaggi tra le estensioni e i browser dato che la loro comunicazione può coinvolgere anche i cookie del browser.

Safari invece, nel 2010 come molti altri browser, decise di implementare uno standard proprietario similare alle estensioni di Chrome a livello a livello progettuale programmabili tramite il menù sviluppatori di Safari [54]. Tuttavia a partire dal 2018 Safari smise di supportare tale da modello adottando solo ed esclusivamente quello delle Safari App Extension, che richiesero obbligatoriamente XCode per la loro progettazione [55].

Safari al momento è l'unico browser ad adottare un modello differente da quello delle estensioni di Chrome, escludendo Firefox che, seppur supportandole, consiglia l'adozione del modello proposto dal W3C. In un annuncio di giugno

2020 Apple, a partire da MacOS 11 tramite Safari 14, decide di supportare l'API WebExtension, tuttavia per renderle disponibili tramite il Mac App Store è necessario creare un progetto ad hoc tramite XCode 12 [56].

Capitolo 4
Architetture a confronto: Chrome vs. Safari

Conclusa la panoramica sui browser più utilizzati attualmente, ci focalizziamo sui due browser maggiormente coinvolti dall'argomento della tesi, Chrome e Safari, enunciando in particolare le loro differenze a livello di architettura e di sviluppo.

4.1 Visione di insieme

Safari e Chrome sono due browser importanti per i loro contributi alla storia dei browser, specialmente riguardo l'adesione ai protocolli W3C e nell'ottimizzazione per motivi differenti. Per sottolineare la loro importanza a livello storico, basti pensare che il motore HTML di Safari, WebKit, fu utilizzato da moltissimi browser grazie alle sue caratteristiche tali da renderlo ottimo come motore di rendering e anche molto facile da integrare in un applicazione, rendendo gran parte della concorrenza basata su questo motore. Lo stesso Google Chrome, nelle prime versioni, fece utilizzo di WebKit, seppur modificato per ospitare il motore Javascript V8 al posto di JavaScriptCore.

Google Chrome ha una notevole influenza riguardo lo sviluppo dei browser grazie ai ritmi e regolarità di aggiornamento, le estensioni facili da progettare, il motore Javascript V8 estremamente efficiente grazie alla compilazione JIT permettendo scrittura di script Javascript assai più elaborati e, in tempi successivi, grazie alla fork Blink basata su WebKit, permettendo modifiche anche sostanziali al motore HTML originario.

Ereditando da WebKit la facile riusabilità del motore all'interno di altre applicazioni, molti browser aventi in tempi precedenti motori proprietari, decisero di basare le loro applicazioni su Blink, questo principalmente a causa delle prestazioni di quest'ultimo ottimizzate per i processori multicore parallelizzando le operazioni.

Entrambi i browser sostengono una filosofia open source con alcune differenze: Chrome ha una licenza proprietaria, ma Google offre un browser totalmente open source denominato Chromium supervisionato dal Chromium Project, dal quale nascono non solo i nuovi browser basati su Blink, ma anche nuove funzionalità che possono essere implementate in un secondo momento dai browser basati su questo software. Safari invece offre tramite licenza open source solo WebKit, il framework che comprende il motore HTML WebCore e il motore Javascript JavaScriptCore; tutto il resto dell'applicazione è coperto da licenza proprietaria.

Infine, i due browser hanno una visione differente riguardo gli aggiornamenti: Chrome viene aggiornato molto frequentemente e distingue nettamente i suoi stadi di sviluppo in quattro canali, ispirando anche in questo caso gli altri browser basati su Chromium; Safari invece, a meno di bugfix importanti, tende a essere aggiornato solo due volte all'anno, seppur Safari Technology Preview, la versione destinata agli sviluppatori, possa contare su aggiornamenti più frequenti.

4.2 Processo di gestione dell'evoluzione sw

Safari, per la versione offerta al pubblico, mantiene tuttora, eccetto casi particolari riguardanti importanti carenze di sicurezza, una filosofia vecchio stampo riguardo gli aggiornamenti e l'evoluzione del software, in particolare negli ultimi tempi tende a rilasciare solo due major release all'anno, una a settembre che introduce la

nuova versione corredata di nuove funzionalità e una a marzo che solitamente mira maggiormente alla sicurezza oppure al supporto di nuove API di successo per la programmazione web [57].

Tuttavia Apple offre anche Safari Technology Preview, una versione di Safari aggiornata assai più frequentemente destinata principalmente agli sviluppatori, seppur possa essere utilizzata anche dagli utenti: Safari Technology Preview offre allo stato embrionale nuove funzionalità come il supporto agli ultimi standard del W3C, il supporto a nuove versioni di WebKit e altre funzionalità che anticipano le novità disponibili nelle versioni stabili di Safari. Coloro che usano questo software possono testare le novità e offrire il loro feedback riguardo bug o ulteriori miglioramenti riguardanti le nuove feature.

Safari è un programma in gran parte closed-source; infatti, l'unica componente contenente codice open-source è WebKit: nelle componenti WebCore e JavaScriptCore, derivate rispettivamente da KHTML e KJS della comunità KDE, la licenza è LGPL 2.1, mentre nelle componenti rimanenti di WebKit la licenza è BSD a 2 clausole [58].

Chrome invece tende ad avere aggiornamenti molto frequenti, in questo caso nettamente divisi tra major release, pubblicate ogni sei settimane, e minor release pubblicate ogni due o tre settimane; questi periodi di tempo sono riferiti alla versione stabile disponibile al pubblico [59]. Infatti Chrome definisce in maniera chiara anche gli stadi di sviluppo del suo software tramite 4 canali: Stable, Beta, Dev e Canary.

Stable è la versione destinata per l'uso quotidiano e per l'utente interessato solo ed esclusivamente alla navigazione nei siti web, essendo questo canale quello maggiormente convalidato dal team di sviluppo e quindi quello meno soggetto ai bug. Beta è un canale che offre in anticipo dalle quattro alle sei settimane agli utenti interessati le nuove versioni con le loro novità; lo scopo di questa scelta è avere maggiori possibilità di incontrare bug grazie alla vasta comunità che, usufruendo della Beta, può comunicare al team di sviluppo errori non riscontrati nei canali Dev e Canary a causa della loro comunità assai più ridotta.

Il canale Dev è uno dei due canali, assieme a Canary, che non garantisce al 100% la stabilità del software ed è offerto agli interessati con un anticipo di tempo che va

dalle nove alle dodici settimane; questo canale serve essenzialmente agli sviluppatori che vogliono adeguarsi velocemente alle ultime API e funzionalità offerte da Chrome. Infine Canary è il canale dedicato allo sviluppo delle future versioni di Chrome e non è destinato all'uso quotidiano a causa non solo della sua instabilità, ma anche per gli aggiornamenti assai frequenti che possono aggiungere o eliminare funzionalità al loro stadio primordiale.

La licenza di Google Chrome è proprietaria, quindi il codice sorgente del software è closed-source; tuttavia è disponibile una versione open-source, Chromium, coperta da licenza libera da cui derivano, oltre Chrome, anche altri browser come le versioni più recenti di Opera e Microsoft Edge. Il codice sorgente è gestito dalla comunità Chromium Project che provvede non solo a offrire il codice sorgente, ma anche numerose guide rivolte agli utenti interessati a contribuire sul suo sviluppo e debugging.

Di conseguenza il motore HTML Blink e il motore Javascript V8, altre componenti correlate con Chromium, sono coperte da licenza BSD rendendoli software open-source, offrendo agli interessati guide dettagliate sulle loro funzionalità e come integrare questi motori in un'applicazione [60].

Chromium vanta alcune differenze rispetto a Chrome: in primis, essendo totalmente open source, di default non supporta codec audio e video coperti da licenza proprietaria come gli MP3 o i video con codec H.264, supportando solo quelli coperti da licenza libera; il supporto di questi formati è possibile solo tramite l'installazione di plugin esterni.

Inoltre Chromium, anche a causa di un attività molto intensa da parte della comunità nell'aggiornare il software, non possiede una funzionalità di aggiornamento automatico a differenza di Chrome; quest'ultimo aggiunge ulteriori funzionalità proprietarie come la visualizzazione di PDF, il supporto di codec proprietari, restrizioni sull'installazione di estensioni riguardo la loro provenienza e, a differenza di Chromium, raccoglie i dati dell'utente dove possono figurare informazioni riguardo il crash dell'applicazione o le statistiche di utilizzo inviandole ai server di Google [61]. Nella tabella qua sotto verranno illustrate le principali differenze tra Chrome e Chromium.

	Chromium	Chrome
Installazione	Laboriosa e complicata a causa della mancanza di un eseguibile che automatizzi la configurazione del browser. La compilazione del suo codice sorgente richiede una buona conoscenza del prompt dei comandi ed è estremamente dipendente dal sistema operativo sul quale si desidera configurare l'applicazione.	Molto semplice: dal sito di Google si scarica un eseguibile che configura il programma in modo da utilizzare il browser subito dopo il termine dell'installazione. Il sito offre l'eseguibile opportuno in base al sistema operativo dell'utente che desidera utilizzare Chrome.
Aggiornamenti	Non ha un sistema di aggiornamento automatico. Il ritmo degli aggiornamenti di Chromium è molto elevato.	Google offre 4 canali di sviluppo dai ritmi di aggiornamento differenti. La versione Stable, destinata per l'uso quotidiano, riceve major release in lassi di tempo che vanno dalle 4 alle sei settimane. Il browser offre un sistema di aggiornamento automatico.
Supporto dei file multimediali	Supporta solo codec coperti da licenza libera: WebM, Theora, Ogg Vorbis, Opus, WAV, VP8 e VP9.	Oltre ai codec supportati da Chromium, il browser di Google supporta codec coperti da licenza proprietaria: MP3, AAC, H.264
Funzionalità proprietarie	Non contiene funzionalità proprietarie. Per integrare	Rispetto a Chromium aggiunge funzionalità

	funzionalità aggiuntive come il supporto a codec proprietari è necessaria l'installazione di plugin esterni.	proprietarie: visualizzazione dei PDF, un sistema di aggiornamento automatico e il supporto ai codec proprietari.
Privacy e sicurezza	È un browser poco sicuro a causa della poca stabilità, rendendo l'applicazione più soggetta a bug e crash, e per la mancanza di un sistema di aggiornamento automatico. Rispetto a Chrome raccoglie e invia meno informazioni; non invia informazioni riguardo crash dell'applicazione e statistiche di utilizzo.	Chrome è un browser, nella versione Stable, molto stabile grazie al debugging rigoroso compiuto tramite i canali Beta, Dev e Canary. Il browser raccoglie informazioni inviandole ai server Google riguardanti principalmente il sistema operativo, la versione del browser e informazioni su eventuali crash. Il sistema di aggiornamento automatico fornisce maggiore sicurezza.
Pubblico a cui l'applicazione è destinata	Principalmente indirizzato per gli sviluppatori di browser: essi possono contribuire al miglioramento del codice sorgente di Chromium o progettare un nuovo browser basandosi sul codice disponibile grazie alla licenza libera.	Dipendente dal canale di sviluppo considerato: Stable offre un browser stabile pensato per l'utilizzo quotidiano. Gli sviluppatori di siti web possono usufruire di Beta, Dev e Canary per aggiornare i loro siti alle versioni più recenti di Chromium, mentre gli sviluppatori di browser possono usufruire di questi tre canali per verificare la loro stabilità.

4.3 Il motore HTML

I motori HTML dei due browser, componenti software che si preoccupano di leggere il linguaggio di markup HTML e le sue risorse in rappresentazioni grafiche interattive, hanno contribuito molto sulla storia dei browser e l'evoluzione dei siti web man mano sempre più complessi a livello grafico e di interazione: Safari è basato su WebCore, che assieme al motore Javascript JavaScriptCore forma il framework WebKit, mentre il motore HTML di Chrome, allo stato attuale, è basato su Blink [62].

WebCore, introdotto nel 2003 assieme alle prime versioni di Safari, è una fork del motore di rendering KHTML, pensato per aderire il più fedelmente possibile alle specifiche DOM richieste dal W3C, oltre a essere uno dei primi a supportare correttamente la scrittura bidirezionale [63]. Apple applicò notevoli migliorie nel rendering del codice HTML, condividendo le patch con KDE, la comunità che sviluppò KHTML; tra i traguardi di rilievo del motore WebCore, esso rese Safari il primo browser a superare appieno Acid2 [64].

Acid2 è un test programmato per mettere alla prova la conformità dei browser agli standard HTML, CSS 2.1, immagini PNG e ai data URI scheme. Altro traguardo di rilievo fu il superamento del test Acid3, basato prevalentemente sugli standard DOM e ECMAScript [65]; anche qua WebCore fu uno dei primi motori a superare il test assieme a Presto, il motore HTML di Opera; questi traguardi resero WebCore, e di conseguenza WebKit, di enorme successo, al punto tale che pure Chrome, per le sue prime versione, usò questo motore HTML [66].

A partire dal 2010, WebKit si evolse in WebKit2 [67], un framework che separa in modo ben definito i compiti da assegnare a ogni processo, dove ognuno di essi ha uno scopo specifico (uno è dedicato all'interfaccia grafica, un altro al rendering del codice HTML eccetera), con vari benefici a livello di robustezza (l'interfaccia e il rendering sono totalmente separati a livello di architettura) e di prestazioni nelle CPU multicore.

Per permettere la visualizzazione delle risorse HTML di un sito web, WebCore comincia a renderizzare l'albero di nodi rappresentante il DOM del documento

creando un albero di renderer, dove ogni nodo renderer rappresenta un elemento specifico, come i singoli tag o la presenza di documenti SVG, tramite una procedura top-down ricorsiva (dalla radice ai figli) detta attachment.

Tramite l'attachment WebCore analizza anche le query CSS per ottenere ulteriori informazioni sulla renderizzazione dei nodi e le collega con questi ultimi in modo da mantenere le informazioni e eventualmente aggiornare quando vengono modificate le regole CSS. Dopo quest'operazione, su tutti i nodi dell'albero WebCore si preoccupa di determinare la loro posizione e la loro dimensione tramite una procedura ricorsiva definita layout che parte dalla radice dell'albero [68].

Il motore HTML Blink, pubblicato nel 2013 e basato su WebKit, è pensato per facilitare le modifiche sul suo codice sorgente perchè Google implementa in maniera differente funzionalità come l'esecuzione multicore delle operazioni. Inoltre, l'utilizzo di una fork rende assai più facile lo sviluppo di idee innovative o rivoluzionare l'architettura del motore di rendering [69].

Il funzionamento di Blink è strettamente legato a Chromium ed entrambi sono basati su un'architettura multiprocesso elaborata, infatti, seppur teoricamente ogni pagina web ha a sua disposizione un processo tutto per sé, nella realtà capita spesso che più pagine o iframes condividano un unico processo, spesso a causa della poca memoria RAM o di troppe pagine aperte.

D'altra parte, Blink, seppur abbia un thread principale con cui compie la maggioranza delle operazioni, può creare nuovi thread relativi a script destinati a rimanere eseguiti per molto tempo come i Web Workers. Inoltre, Blink può legarsi strettamente al motore Javascript V8 per supportare i content script delle estensioni di Chrome, script pensati per modificare i contenuti delle pagine web visitate dall'utente, seppur essi vengano isolati dagli script del sito web visitato per motivi di sicurezza [70].

4.4 Il motore Javascript

Sia Chrome che Safari contribuirono molto pure col loro motore Javascript, apportando notevoli migliorie nell'esecuzione del codice; il motore Javascript di

Chrome è V8, mentre Safari ha all'interno di WebKit JavaScriptCore (una fork di KJS).

V8, introdotto nel 2008 grazie a Chrome e progettato da Lars Bak, è un interprete Javascript totalmente differente da quelli sviluppati fino a quel momento dai concorrenti. Infatti, prima del rilascio di V8, i motori Javascript implementati in tali applicazioni erano dei semplici interpreti totalmente privi di processi di compilazione.

Se l'interpretazione, da una parte, facilita il processo di debugging dell'interprete, dall'altra ciò si ripercuote notevolmente sulle sue prestazioni sia a livello temporale a causa della traduzione e identificazione "in tempo reale" delle istruzioni scritte dal programmatore e per l'identificazione e accesso alle variabili scritte nelle istruzioni che richiedono ripetutamente l'accesso alla memoria a run-time, sia sulla memoria occupata a causa dell'overhead dato dall'interprete stesso [71].

Questo problema si notò in particolare quando nel 2004 venne introdotto Google Maps, uno dei primi siti che fa ampio utilizzo di codice Javascript dimostrando le notevoli proprietà del linguaggio, tali da mostrare una mappa del mondo con un numero di dettagli incredibile a livello di quantità di informazioni e dinamico a seconda del livello di zoom applicato dall'utente [72].

V8 è il primo motore Javascript implementato in un browser a fare utilizzo della compilazione just-in-time, o JIT, permettendo al motore di compilare il codice Javascript in bytecode e, in certi casi, codice macchina durante la sua esecuzione, rendendo quest'ultima assai più efficiente a livello di tempo rispetto alla concorrenza; ciò è uno dei motivi dell'enorme successo di Chrome portando i browser concorrenti ad adottare anch'essi la compilazione just-in-time [73]. Inoltre, le prestazioni di V8 gli ha permesso un certo successo anche in ambito server, spesso eseguito assieme al framework Node.js.

V8 funziona nel seguente modo: reperisce il codice sorgente, lo analizza tramite uno scanner e un parser in modo da generare un albero sintattico astratto (o Abstract Syntax Tree o AST) dove ogni nodo identifica un costrutto ben conosciuto del linguaggio, il suo interprete Ignition genera bytecode tramite l'albero che poi esegue tramite interpretazione: già questi passaggi contribuiscono

molto sulle performance perchè l'utilizzo del bytecode semplifica la traduzione e, di conseguenza, l'esecuzione delle istruzioni mantenendo le medesime informazioni del codice sorgente.

Nel caso V8 identifichi tramite delle euristiche, ad esempio, una funzione utilizzata molto frequentemente, esso può utilizzare un compilatore, Turbofan, per ottimizzare ulteriormente il bytecode in codice macchina vero e proprio, velocizzando notevolmente l'esecuzione; Turbofan può anche correggere la sua compilazione nel caso in cui il codice macchina prodotto non corrisponda semanticamente col codice sorgente a causa di euristiche errate: in tal caso, elimina il codice macchina prodotto, si continua l'esecuzione tramite il bytecode e nel mentre Turbofan compie una nuova compilazione con le nuove informazioni a disposizione [74].

JavaScriptCore, il motore Javascript di Safari e introdotto nel 2003, è una fork del motore KJS; come gli altri motori Javascript dell'epoca, JavaScriptCore era un interprete privo di processi di compilazione ma già nel 2008 JavaScriptCore fa utilizzo della compilazione just-in-time, dimostrando assieme a V8 le potenzialità a livello di performance della compilazione just-in-time.

A dispetto dei tanti nomi affibbiatogli nel tempo come SquirrelFish, Nitro, FTL e altri ancora, la documentazione ufficiale si riferisce a lui solo col nome JavaScriptCore. Esso è composto di vari componenti [75], similari nei loro compiti alle componenti di V8:

- Lexer, dedicato all'analisi lessicale del codice generando dei token.
- Un parser, che genera un albero sintattico, scritto in bytecode, in base ai token generati tramite il lexer.
- L'interprete LLInt (Low Level Interpreter) che esegue il bytecode generato dal parser. LLInt è pensato per avere un costo iniziale molto ridotto, è scritto in un linguaggio assembly denominato offlineasm.
- Baseline JIT, DFG JIT e FTL JIT sono tre compilatori che vengono utilizzati quando si devono compilare funzioni o cicli utilizzati molto spesso; la scelta del compilatore dipende dalle euristiche calcolate. Baseline JIT è dedicato alle ripetizioni che occupano poco spazio in memoria e assieme a LLInt può

informare DFG JIT di possibili ottimizzazioni del codice. DFG JIT è un compilatore assai più potente in grado di ottimizzare notevolmente il codice, ricoprendo un ruolo simile a quello di Turbofan di V8 e, come esso, se incontra problemi durante l'esecuzione a causa di un euristica errata, applica una de-ottimizzazione dando un segnale al Baseline JIT dove quest'ultimo continuerà l'esecuzione. FTL JIT è molto simile a DFG JIT, viene invocato in particolare per le operazioni molto ripetitive come funzioni invocate migliaia di volte o cicli ripetuti decine di migliaia di volte.

4.5 I plug in e le estensioni

Le estensioni per browser sono programmi che aggiungono funzionalità ai browser o migliorano quelle già presenti; alcune di esse furono così influenti da rendere le loro funzionalità oggigiorno presenti di default nei browser odierni. Se le estensioni, tempo fa, avevano metodi differenti di progettazione a seconda del browser interessato, negli ultimi anni la progettazione delle estensioni per Google Chrome è ormai supportata anche da altri browser come Microsoft Edge e Mozilla Firefox.

Tra i browser più utilizzati, solo Safari continuò, fino al 2020, a usare un modello proprietario differente di progettazione delle estensioni, le Safari App Extension. Adattare un'estensione di Chrome per Safari, o viceversa, non è una procedura banale, dato che hanno pochissimi punti in comune tra di loro rendendo necessaria, in buona parte dei casi, la progettazione da zero di una nuova estensione.

Le estensioni di Chrome, introdotte nel settembre 2009 dopo una lunga fase di beta testing [76], vennero pensate per essere facilmente programmabili da qualunque programmatore che conoscesse i linguaggi HTML, CSS e Javascript, corrispondenti alle basi gli dello sviluppo di un sito web; ciò, assieme a vari esempi forniti direttamente da Google e a una documentazione completa, permise alle estensioni di essere più facili da programmare grazie alla presenza di guide fin dal loro rilascio e di essere sviluppate da chiunque avesse un account Google, la cui registrazione è gratuita.

Ciò si riflette sul loro enorme successo: a gennaio 2010 furono disponibili già 1500 estensioni tramite Chrome Web Store garantendo già in poco tempo un catalogo molto vasto che potesse soddisfare gli utenti, mentre nel Giugno 2012 furono 750 milioni le installazioni di estensioni provenienti dal Chrome Web Store. Tale facilità di programmazione aiutò la pubblicazione delle estensioni e di conseguenza una navigazione fortemente personalizzata e fu uno dei vari motivi che resero Chrome il browser più utilizzato superando Internet Explorer e la concorrenza [77].

Sempre nel 2012, tramite un aggiornamento, Chrome aggiornò l'API delle estensioni alla seconda versione [78], differente in particolar modo per l'implementazione non più obbligatoria dell'interfaccia [79]. A partire dal 2020 Chrome cominciò a supportare la terza versione delle estensioni di Chrome, molto differente dalle precedenti e maggiormente mirata alle performance e alla privacy [80].

Le Safari App Extension, rese inizialmente disponibili per MacOS Sierra e El Captain con Safari 10 e annunciate tramite il WWDC del 2016, vennero pensate per estendere la programmazione delle applicazioni Mac unendo la progettazione delle estensioni con la scrittura di codice in linguaggio Swift, linguaggio di programmazione pensato per sostituire le applicazioni allora programmate in Objective-C.

Le Safari App Extension sono basate sul concetto delle App Extension, applicazioni che consentono all'utente di accedere alle loro funzionalità tramite MacOS e pensate per uno scambio di messaggi e risorse particolarmente elaborato tra le applicazioni. Ciò rende le Safari App Extension, programmate in codice nativo Swift e sviluppate tramite Xcode, più facili da sviluppare da parte dei programmatori abituati a usare l'IDE e esperti delle API offerte da esso e più consci delle potenzialità del linguaggio di programmazione Swift all'interno di queste applicazioni.

Le Safari App Extension vengono distribuite nel Mac App Store in un unico pacchetto comprendente sia l'applicazione adibita come installer che l'estensione vera e propria da aggiungere a Safari, non rendendo necessario all'utente scaricare separatamente l'estensione; questa strategia di distribuzione aiuta la comunicazione tra l'applicazione nativa e l'estensione creando una forte simbiosi, permettendo a

entrambe le applicazioni, estensione e installer, di avere la medesima versione essendo distribuite assieme riducendo le incompatibilità tra loro.

Le principali tipologie di estensioni programmabili per Safari portano con sè delle novità: ad esempio, se si intende di progettare un content blocker (applicazione che blocca o nasconde elementi di una pagina), anziché eseguire l'operazione a runtime, con le Safari App Extension si può dichiarare dal principio quali contenuti nascondere o bloccare in maniera efficiente grazie all'utilizzo di una libreria nativa offerta da XCode, WebKit, oppure rendere un content blocker per iOS disponibile anche per MacOS grazie all'utilizzo delle medesime API, automatizzando la conversione di un estensione iOS a una MacOS evitando al programmatore la riscrittura manuale del codice.

Altre novità delle Safari App Extension riguardano la modifica delle pagine web dato che, di default, sono progettate per modificare solo alcune pagine. Nel caso, invece, l'estensione applichi le modifiche su ogni pagina, l'utente che ne usufruisce viene avvisato dei possibili rischi di sicurezza.

Molta importanza, per le estensioni di Safari, è data dalla loro certificazione: infatti l'utente, di default, può installare solo ed esclusivamente le estensioni distribuite tramite la licenza di programmazione Apple enfatizzando sulla sicurezza; altrettanto importante per Apple è il riutilizzo del codice nativo e delle sue libreria velocizzando l'adattamento di applicazioni preesistenti alla comunicazione con le estensioni [81].

Capitolo 5
Polymorph: un convertitore per estensioni per Safari

Polymorph è un applicazione realizzata tramite il framework Node.js, pensata per aiutare lo sviluppatore di estensioni ad adattare i suoi progetti alle differenze implementative presenti nei browser riguardo le estensioni.

Nello specifico, è pensata per adattare le estensioni di Chrome a Safari, rendendole Safari App Extension, e le Safari App Extension rendendole delle Chrome Extension aderenti allo standard Manifest V2 rendendole compatibili coi browser basati su Chromium. Il fatto che il software sia programmato in Javascript, inoltre, permette una migliore comprensione del suo codice per i progettisti di estensioni, dove Javascript ricopre una parte rilevante di queste applicazioni.

Polymorph si presenta con un'interfaccia piuttosto semplice: nella porzione superiore si selezionano la directory di input, quella che contiene l'estensione che il progettista desidera convertire, e la directory di output, in modo che l'estensione convertita venga aggiunta nel percorso desiderato. Non c'è il rischio di una sovrascrizione del progetto originale: infatti, a prescindere dal percorso di output considerato, la cartella contenente l'estensione convertita avrà un nome differente da quello del progetto originale.

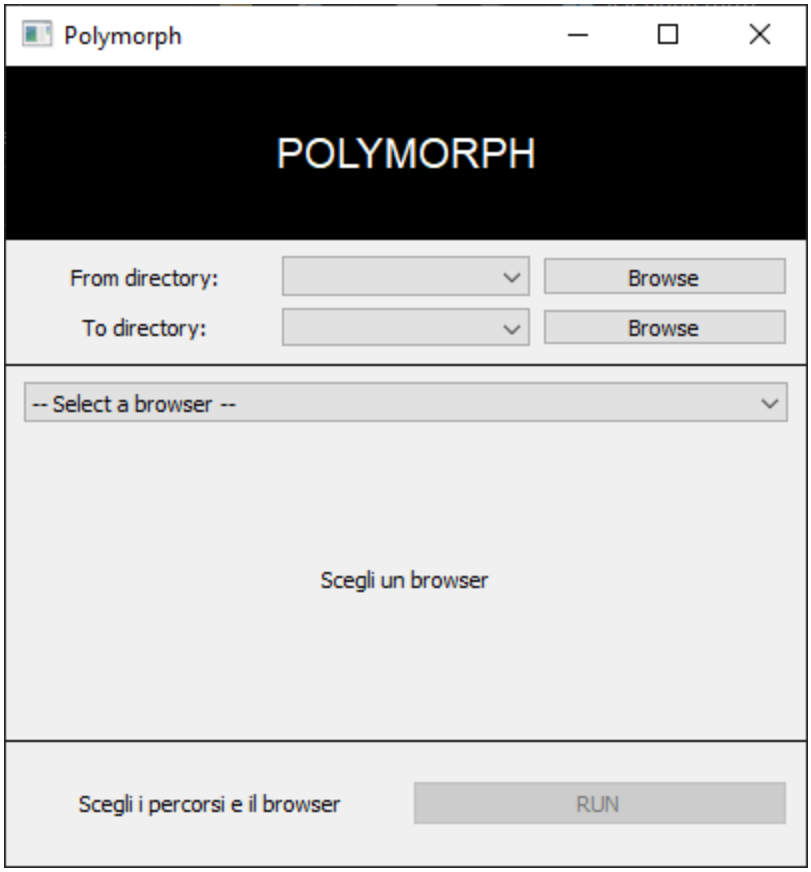
Sotto queste due opzioni si può scegliere lo standard a cui adattare l'estensione selezionata; scegliere un'opzione farà comparire dinamicamente un testo che comunica quali browser possano usare lo standard. Allo stato attuale ci sono due possibili scelte: "Chromium", col quale adattare l'estensione ai browser che supportano Chromium (Chrome, Edge eccetera) o "Safari 13 (MacOs High Sierra)", per realizzare una Safari App Extension.

Infine c'è un bottone con cui far partire la conversione, a patto che tutte le impostazioni non siano vuote o non valide; in tal caso, accanto al bottone apparirà un testo che comunica le problematiche o mancanze pregando l'utente di riempire i campi vuoti o, riguardo il percorso di input, inserirne uno contenente un estensione. Il medesimo testo avvisa l'utente quando Polymorph ha finito di effettuare la conversione.

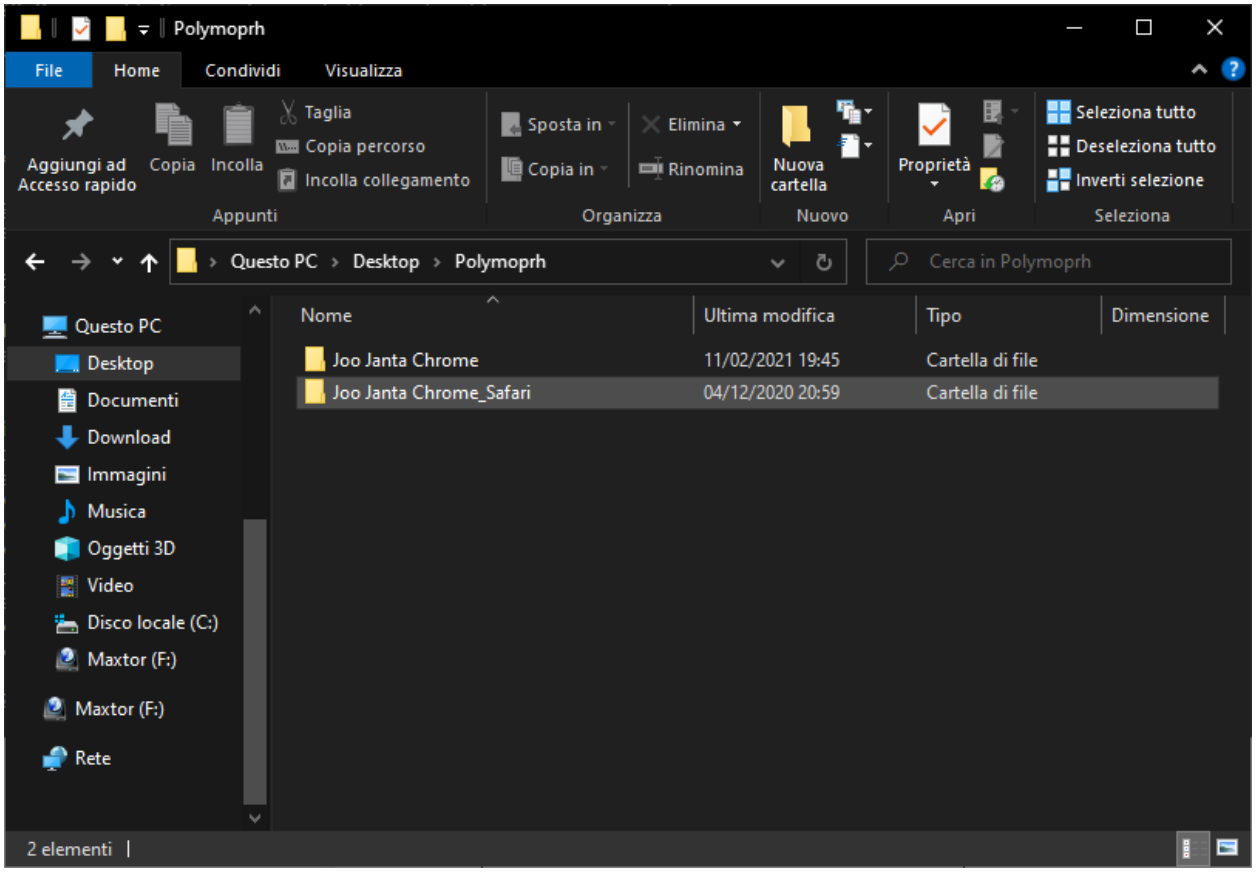
Assieme al software Polymorph è presente un documento di istruzioni che aiuta il programmatore nel completare la progettazione dell'estensione appena convertita, sia per le Safari App Extension che per le estensioni destinate per Chrome poiché la procedura non è totalmente automatizzata e alcuni passaggi richiedono l'intervento del programmatore, che vanno dalle impostazioni o all'aggiunta di librerie (passaggi necessari per le Safari App Extension, eseguibili solo tramite Xcode) a errori logici dipendenti dai differenti motori Javascript.

Polymorph è un programma scritto con linguaggio Javascript e fa utilizzo, grazie alla libreria Nodegui, del framework grafico Qt; la loro scelta non è casuale dato che Javascript è un linguaggio molto usato nella programmazione web e per la programmazione di estensioni, Qt invece, oltre a essere multipiattaforma, rende particolarmente facile la costruzione di interfacce elaborate grazie a una struttura a scatole.

Questa costruzione delle interfacce, al di là delle differenze implementative, è un concetto familiare ai progettisti di siti web essendo questo concetto presente anche nel linguaggio HTML. Sia Javascript che Qt che Nodegui sono ben documentati, facilitando la comprensione del codice sorgente di Polymorph e la possibilità di estenderlo e migliorarlo.



L’interfaccia del programma. Essa è divisa in tre sezioni (escludendo il titolo), rendendola intuibile anche a coloro che non conoscono il compito principale per cui questa applicazione è stata progettata.



Polymorph, per evitare di sovrascrivere il contenuto originale, pubblica il suo risultato in un'altra cartella, che prende il nome della cartella di input aggiungendole un "_Safari" o "_Chromium" a seconda del browser su cui si desidera progettare l'estensione.

5.1 Struttura del pacchetto di estensioni x Chrome

Le estensioni per Chrome sono composte principalmente di cinque file: un file manifest.json che imposta le autorizzazioni, un file HTML col ruolo di interfaccia allegata all'interno di un popup invocato quando si clicca sulla sua icona, un eventuale file CSS che può arricchire l'interfaccia oppure essere utilizzato dallo script esterno, e due script, uno interno, il background script, legato all'interfaccia dell'estensione, e di uno script esterno, il content script, con la funzione di modificare le pagine web visitate dal browser [82].

Questa struttura, tuttavia, non è l'unica: infatti un estensione per Chrome può anche non avere nessuna interfaccia oppure averne una che occupa un tab del browser, soluzione usata principalmente per regolare alcune sue impostazioni avanzate. L'unico file fondamentale di queste estensioni è il manifest.json contenente i permessi e i file coinvolti, essenziale per renderle riconoscibili al browser.

Oltre al manifest.json e agli elementi precedentemente citati, un'estensione di Chrome può avere altri elementi facoltativi che sono:

- Font personalizzati.
- L'icona.
- Sidebar, pop-up e pagine di impostazioni.
- Risorse accessibili via web.

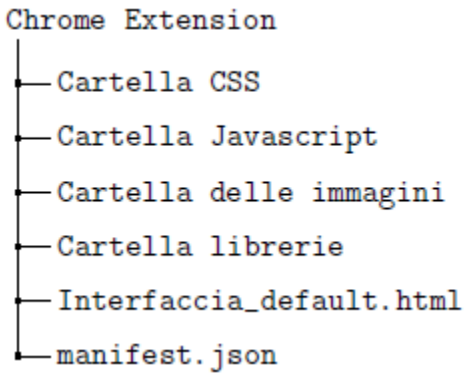
Quindi le estensioni progettate per Chrome possono essere molto differenti tra loro, seppur il modello citato inizialmente pare essere il tipo di estensione più ricorrente. La progettazione di estensioni per Chrome è molto ben documentata e vanta guide esaustive anche nel sito ufficiale di Google che riguardano non solo la progettazione di un esempio di estensione, ma anche spiegazioni dettagliate di ogni istruzione implementabile.

Questa minuziosità della documentazione rende lo sviluppatore conscio dei contributi di ogni riga di codice facilitando la comprensione del loro codice non solo da parte del progettista, ma anche da coloro intenzionati a osservare il codice sorgente delle estensioni che decisero di installare su Chrome, questo grazie anche a una gestione avanzata delle estensioni tale da poter esplorare la loro struttura di progetto con la possibilità di osservare con il browser stesso il codice dei file [83].

Google offre anche dei consigli di buona programmazione delle estensioni [84]: in particolar modo, sconsiglia l'inserimento di pubblicità nelle pagine web, le estensioni con troppe funzionalità non correlate tra loro (è preferibile progettare estensioni separate ognuna con uno scopo ben specificato, ciò viene definito single-purpose policy) o l'inserimento di elementi unito ad altre funzionalità.

Google consiglia allo sviluppatore di inserire solo le autorizzazioni strettamente necessarie per il funzionamento dell'estensione in modo da evitare violazioni della policy e fare modifiche alle pagine web con un unico scopo ben specificato. Se lo sviluppatore rispetta le policy delle estensioni di Chrome può distribuire la sua estensione nel Chrome Web Store, che garantisce agli utenti estensioni più sicure rispetto a siti di terze parti [85].

Le estensioni di Chrome, teoricamente, non hanno regole rigorose sull'organizzazione dei file di progetto, essendo l'unico elemento obbligatorio di esse il manifest.json; tuttavia, per avere i file di progetto ben organizzati, si tende a suddividerli in base alle loro funzionalità. Questo è un esempio di buona organizzazione dei file in un progetto pensato per programmare un estensione di Chrome:



Perchè questa organizzazione è buona? Per i motivi seguenti:

- Ogni cartella rappresenta uno scopo ben specificato, che va dal markup allo scripting, rendendo chiaro lo scopo dei file anche solo osservando la struttura del progetto.
- È buona cosa tenere la cartella delle librerie separata da quella degli script; infatti, nel caso in cui un content script debba usare le funzionalità di una libreria, bisogna aggiungere nel manifest.json l'autorizzazione di modificare i siti web a entrambi a file seguendo un ordine di dichiarazione dei permessi rigoroso, ovvero prima si devono assegnare i permessi alle librerie e dopo agli script che utilizzano queste ultime; non seguire questo ordine porta a problemi di debugging. Inoltre, separare le librerie dagli script aiuta anche a

distinguere quali script siano librerie e quali no, rendendo più semplice anche il riutilizzo di queste librerie in altri progetti.

- L’elenco mostrato è piuttosto semplice; tuttavia, se l’estensione è composta di molti file, possono esserci più cartelle, ad esempio può esserci una cartella contenente font personalizzati all’interno della cartella coi file CSS, si possono distinguere tramite cartelle le librerie usate solo ed esclusivamente per l’interfaccia o per il content script, oppure creare una cartella di file HTML nel caso vengano utilizzati per scopi particolari come, per esempio, opzioni avanzate per l’estensione o una sua cronologia; solitamente questi file HTML aggiuntivi sono progettati per occupare un tab del browser.

Quindi poter organizzare bene i file di progetto è un buon modo per ottimizzare la comprensione del progetto e l’eventuale aggiunta di nuove funzionalità.

5.2 Struttura del pacchetto di estensioni x Safari

Una Safari App Extension, a differenza di un estensione per Chrome, è assai più articolata, non a caso tutti i file obbligatori necessari per il suo funzionamento vengono creati quando l’utente, aprendo XCode, decide di progettare l’applicazione; il progetto creato da Xcode offre non solo l’estensione da aggiungere a Safari, ma crea anche un'applicazione esterna, un installer con cui aggiungere l’estensione al browser in maniera autorizzata.

Togliere anche solo uno dei file obbligatori comporterà il fallimento della compilazione dato che ognuno di essi ha un suo scopo che va dall’impostazione delle autorizzazioni, al riconoscimento dell’applicazione da parte del sistema operativo e di Safari fino agli file di script veri e propri necessari per le operazioni basilari dell’estensione come lo scambio dei messaggi o l’avvio della sua interfaccia. L’unico file che può essere sostituito è l’immagine dell’icona, a patto però che la nuova immagine sia un PNG e venga fatto un riferimento esplicito a essa tramite il file Info.plist, altrimenti avviene un fallimento della compilazione [86].

Essendo poco numerose le Safari App Extension pubblicate, è difficile identificare una procedura standard per la loro progettazione; il numero limitato di estensioni è

dato in parte dalla documentazione scarsa e anche dalla mancanza di una procedura semplice con cui convertire le vecchie estensioni di Safari, denominate Safari Extension o Legacy Safari Extension, al nuovo modello.

Infatti, nonostante il sito ufficiale riporti l'esistenza di una procedura automatica [87], all'atto pratico questa è inesistente costringendo lo sviluppatore a riscrivere gran parte del codice adattandolo a Swift questo perchè le Safari App Extension sono estremamente differenti dallo standard che Apple adottò precedentemente riguardo le estensioni, essendo le Legacy Safari Extension assai più similari a livello di funzionamento alle estensioni di Chrome differendo, in gran parte dei casi, per la sintassi differente delle istruzioni native [88].

La documentazione scarsa rende la progettazione di queste estensioni piuttosto complicata: seppur venga spiegata l'architettura dell'estensione, il sito ufficiale di Apple non riporta un esempio completo di un'estensione funzionante, non aiuta lo sviluppatore a scrivere buon codice riportando solo la sintassi delle istruzioni fondamentali, approfondendo poco il loro contributo; pure trovare i loro link per documentarsi può essere una procedura poco immediata, dato che in alcuni casi vengono linkate istruzioni non valide, legate al vecchio modello di estensioni.

Per documentarsi meglio per la programmazione di una Safari App Extension è caldamente consigliato conoscere il linguaggio Swift, controllando con attenzione che si leggano le istruzioni provenienti dalla medesima versione di Swift e dal medesimo sistema operativo: Swift, a seconda che si programmi per iOS o MacOS o a seconda della versione può avere sintassi molto differenti, usare oggetti in maniera estremamente differente e pure gli oggetti stessi coinvolti nei progetti possono essere denominati in maniera differente [89].

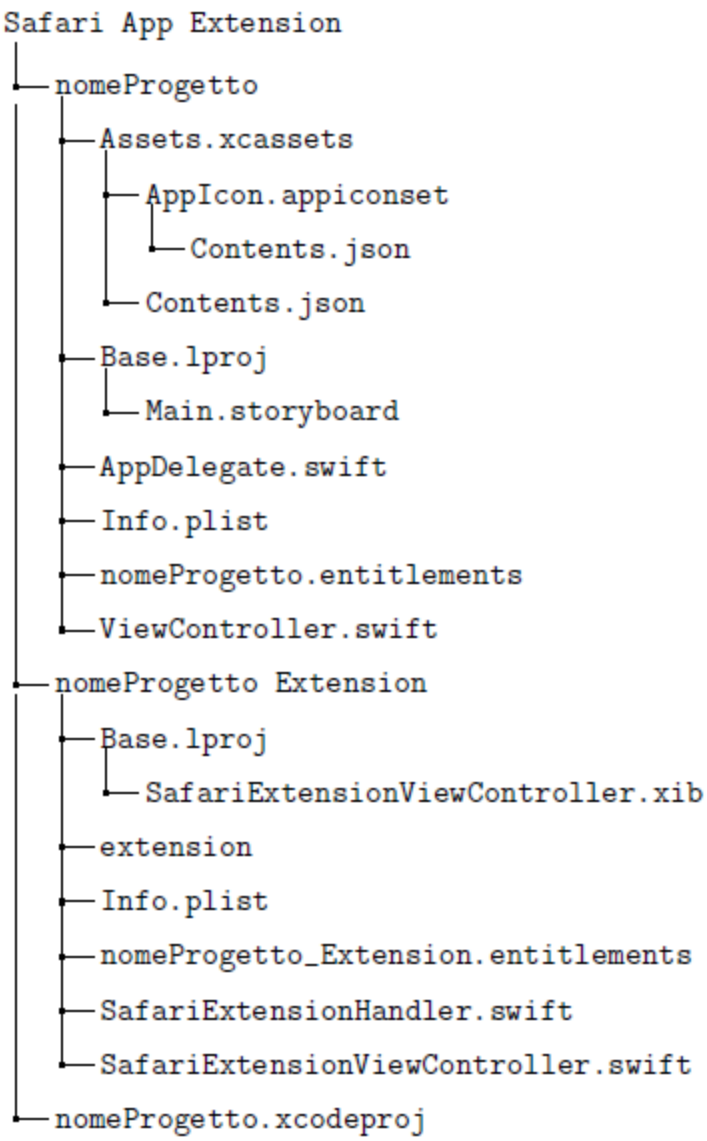
Inoltre, è importante conoscere il concetto di App Extension, modello di programmazione che consente all'applicazione di scambiare messaggi con altri applicazioni all'interno del sistema operativo Apple in modo da personalizzare delle funzionalità o sincronizzare dati e risorse [90]; Apple consiglia caldamente App Extension separate con compiti specifici anzichè un'unica applicazione con tante funzionalità e scopi differenti tramite policy similari a quelle che Google consiglia per le estensioni di Chrome, quindi è buona cosa, nel caso il progettista

sia interessato a implementare molte funzionalità estremamente differenti tra loro, dedicare una singola applicazione a ognuna di esse.

Essendo l'estensione identificata come una vera e propria applicazione, lo sviluppatore, se ha intenzione di progettare una Safari App Extension basandosi su una preesistente per Chrome o Firefox, può programmare l'estensione in modo da contenere una pagina HTML usandola come interfaccia grazie ad apposite librerie, a costo di implementare un secondo scambio di messaggi, in questo caso tra l'interfaccia HTML e l'applicazione che la ospita.

Questo è solo un esempio, ma potenzialmente anche le Safari App Extension possono vantare molte possibilità tra l'integrazione di librerie e scambi di risorse anche con altre applicazioni differenti da Safari, in quest'ultimo caso anche per inviare queste risorse al browser in modo che quest'ultimo possa elaborare le loro informazioni a seconda delle volontà del programmatore [91].

La struttura di una Safari App Extension è la seguente:



Come si nota, una Safari App Extension impone una organizzazione del progetto assai più rigorosa: questo è dovuto al fatto che essa può essere progettata solo ed esclusivamente tramite XCode. La cartella principale è composta di due cartelle, “nomeProgetto” e “nomeProgetto Extension”, e di un eseguibile, “nomeProgetto.xcodeproj”. L’eseguibile permette il caricamento rapido del progetto tramite XCode, aprendo quest’ultimo nel caso non sia già in esecuzione.

Il progetto è nettamente diviso dalle due cartelle poiché ognuna di esse contiene un applicazione distinta. La cartella “nomeProgetto” contiene l’installer dell’estensione, colui che permette l’aggiunta dell’estensione nel browser Safari in modo sicuro e di conseguenza la possibilità di utilizzare l’estensione da parte

dell'utente, senza che vengano modificate le opzioni avanzate del browser legate allo sviluppo di siti e estensioni.

Questa cartella in Polymorph è pressoché identica a quella offerta di default da XCode essendo il suo codice già perfettamente funzionante, tuttavia il programmatore è libero di implementare in questa applicazione funzionalità particolari o elementi che possano beneficiare l'utilizzo dell'estensione, come un testo di istruzioni per il suo utilizzo.

Invece la seconda cartella è quella contenente l'estensione vera e propria, inclusa la sua interazione col browser. La Safari App Extension è basata principalmente su:

- L'interfaccia XIB, determinata da ""SafariExtensionViewController.xib". Questo file è quello con cui il progettista personalizza l'interfaccia dell'estensione con un linguaggio di markup simile al linguaggio XML, adottato anche da altre applicazioni Apple. Il progettista è libero di modificare il codice sorgente oppure impostare l'interfaccia tramite un editor grafico col quale può ottenere una preview del risultato finora costruito. Tuttavia questa preview non può prevedere il comportamento dell'applicazione nel caso vengano aggiunti framework grafici durante l'esecuzione tramite Swift, quindi è comunque necessario analizzare gli script Swift che adempiono a tali compiti.

- Info.plist, il file che raccoglie le autorizzazioni dell'estensione. Anche questo file, come il file XIB legato all'interfaccia, fa utilizzo di una sintassi XML anche se con chiavi proprietarie in modo da distinguerlo all'interno del progetto. La varietà delle autorizzazioni concesse e proibite all'estensione può essere molto vasta, in particolare riguardo il suo comportamento in base alle pagine visitate; un progettista può infatti decidere di applicare le modifiche dell'estensione a qualsiasi sito web oppure personalizzare in maniera più variegata l'azione degli script esterni, definendo per ogni file Javascript in quali URL compiere le modifiche e in quali no.

- SafariExtensionHandler.swift, dedicata ad alcune funzionalità che implicano interazioni tra estensione e browser, come la memorizzazione di nuove pagine visitate in modo da poter applicare le modifiche in un secondo momento.

- SafariExtensionViewController.swift, il file su cui si programmano funzionalità relative alla sua interfaccia e, nel caso vengano integrati framework, scambi di messaggi tra essi e l'applicazione che le ospita, in grado di raccogliere i loro dati per inviarli al browser.

5.3 Problemi particolari e casi speciali

A prescindere dallo standard considerato, che siano le estensioni di Chrome o le Safari App Extension, uno dei passaggi più problematici per la loro progettazione è l'interazione tra il loro script interno, che in buona parte dei casi gestisce la loro interfaccia e/o memorizza i dati, e il loro script esterno, adibito per la modifica delle pagine visitate dall'utente.

In entrambi i casi bisogna non solo conoscere le loro istruzioni native apposite per questa funzione, ma bisogna conoscere anche il concetto dello scambio di messaggi e le regole di una buona programmazione per il message passing, tra cui ridurre il numero di messaggi scambiati al minimo indispensabile. Ridurre al minimo indispensabile il numero di messaggi scambiati, infatti, è vantaggioso in termini di prestazioni essendo il message passing un meccanismo di sincronizzazione tra due applicazioni e, nel caso delle estensioni, questo scambio di messaggi deve essere asincrono a meno di quei casi in cui i dati scambiati siano effettivamente necessari per il proseguimento delle operazioni.

Infatti uno scambio di messaggi sincrono, oltre al deficit prestazionale insito nel metodo di programmazione, aggiunge un ulteriore rallentamento a causa della sua esecuzione perché il mittente di uno scambio di messaggi sincrono rimarrà fermo fino a quando non riceverà una risposta dal destinatario e non è garantito che quest'ultimo dia una risposta immediata. Quindi, quando si parla di scambio dei dati tra gli script delle estensioni, bisogna valutare attentamente:

- Se i dati scambiati siano effettivamente scambiabili solo tramite uno scambio dei messaggi e da elaborare immediatamente (come la ricarica di un tab del browser, l'apertura di una nuova finestra o il risultato di una query avente a che fare coi tab) o se risultano semplicemente dati elaborati dallo script interno memorizzabili all'interno dello storage locale dell'estensione

(booleani, interi, stringhe eccetera). In quest'ultimo caso Chrome, attivando le autorizzazioni opportune, può permettere l'accesso e la modifica della memoria locale dell'estensione sia allo script interno che a quello esterno, velocizzando le operazioni.

- Adottare sempre, nel caso non sia possibile applicare ciò che è scritto nel punto precedente, uno scambio di messaggi asincrono in modo da ottimizzare notevolmente l'esecuzione delle operazioni.
- Nelle Safari App Extension lo scambio di messaggi è obbligatorio per informare lo script esterno non solo delle modifiche da applicare, ma anche di altre informazioni come lo stato di accensione o spegnimento dell'estensione. In questo caso, anziché applicare multipli scambi di messaggi dove ogni scambio trasporta con sé un dato, è buona cosa ridurre tutto ciò in un unico scambio di messaggi tale da portare tutte le informazioni necessarie, ciò è utile anche per semplificare il debugging di invio e ricezione dei messaggi.

Un problema invece esclusivo delle Safari App Extension è la reperibilità dei dati elaborati dallo script interno da parte del content script. In questo caso, dato che bisogna aggiornare il browser degli aggiornamenti riguardo i dati elaborati, si deve ridurre la procedura di aggiornamento delle informazioni al minimo indispensabile, possibilmente questi aggiornamenti devono essere elaborati solo in base alla volontà dell'utente che usufruisce dell'estensione, interagendo con la sua interfaccia.

Ridurre al minimo indispensabile l'aggiornamento di dati della memoria locale permette di ridurre ulteriormente l'utilizzo del message passing e di conseguenza ottimizzare le performance. Per esempio, se l'utente modifica certe impostazioni di un'estensione quando essa è disattivata, è inutile informare lo script esterno degli aggiornamenti essendo stata disattivata; è buona cosa aggiornare lo script esterno solo quando lo stato dell'estensione rimane attivo oppure quando quest'ultimo cambia, da attivo a disattivo o viceversa.

5.4 Gestire una conversione

Tuttavia, in entrambi i casi considerati da Polymorph, ovvero adattare un estensione Chrome a Safari e viceversa, la procedura non è totalmente automatica: è necessario l'intervento del programmatore perché il software non controlla la logica di programmazione, bensì solo la sintassi; quindi alcune istruzioni scritte in maniera legittima possono dare risultati differenti a seconda dell'interprete Javascript utilizzato dai browser; inoltre le Safari App Extension richiedono interventi sui file di progetto per integrare alcune librerie necessarie per il funzionamento dell'estensione convertita.

Progettare un estensione che possa funzionare sia su Chrome che su Safari senza fare alcuna modifica al suo codice non è possibile: infatti buona parte delle estensioni fa utilizzo di uno scambio di messaggi tra script interno e esterno per compiere funzionalità articolate come la modifica di una pagina visitata dal browser o il blocco delle pubblicità. Lo scambio di messaggi fa utilizzo, sia su Chrome che in Safari, di istruzioni native che richiedono sempre una riscrittura se si ha intenzione di pubblicare un estensione proveniente da Chrome sul Mac App Store o viceversa.

Oltre alle modifiche sui file Javascript, è necessario riscrivere i permessi e le autorizzazioni: infatti le estensioni Chrome le memorizzano nel manifest.json con la sintassi JSON, invece i permessi delle Safari App Extension sono memorizzati nel file Info.plist tramite sintassi XML e richiedono una sintassi più rigorosa: ad esempio, se nelle estensioni Chrome non è necessario specificare lo schema degli URL proibiti, d'altro canto nelle Safari App Extension la dichiarazione dello schema degli URL è obbligatoria.

Riguardo la separazione tra frazioni di codice riciclabili e frazioni da riscrivere questo dipende dal modello di progettazione delle estensioni: le Safari App Extension separano già, a livello progettuale, il codice strettamente legato all'estensione e quello che lega l'estensione a Safari.

La memorizzazione dei dati, lo scambio dei messaggi e altre funzioni strettamente legate a Safari sono totalmente scritte in Swift, infatti la porzione di codice Javascript da riscrivere è molto ridotta a causa delle pochissime istruzioni native Javascript legate Safari e le poche presenti riguardano solo ed esclusivamente l'invio e la ricezione dei messaggi.

Nelle estensioni Chrome non esiste una distinzione così netta tra codice riciclabile e codice da riscrivere questo perché Chrome offre molte istruzioni native utili al programmatore e strettamente legate al browser, rendendole valide solo ed esclusivamente su Chrome e sugli altri browser che supportano le sue estensioni.

Inoltre le istruzioni native di Chrome usano molto spesso le callback rendendo il riciclaggio del codice, specie con le callback annidate, poco immediato. Di seguito verranno elencati i passaggi necessari per convertire un estensione, tramite Polymoprh, al modello di Chrome o a quello di Safari, rapportando gli effettivi vantaggi rispetto alla programmazione manuale.

Passare da una Chrome Extension a una Safari App Extension

Passando da un Chrome Extension a una Safari App Extension i passaggi sono numerosi; in primis servono i seguenti requisiti:

- Una versione di XCode a partire dalla 10.1
- Di conseguenza, MacOS High Sierra o uno più recente
- Safari 12 o più recente

Rispettare i requisiti è fondamentale a prescindere dall'utilizzo di Polymorph poichè, non avere anche solo uno di questi elementi, rende il debugging della Safari App Extension impossibile. Successivamente bisogna adattare la sintassi di Javascript al modello richiesto da Safari: se questa procedura, in gran parte, è gestita automaticamente da Polymorph (a meno di possibili errori logici dati dall'interprete differente), manualmente, invece, la procedura è assai laboriosa, poco documentata e poco intuitiva. I passaggi che il progettista deve effettuare sono:

Creare il pattern di progettazione offerto da XCode e imparare a utilizzare i suoi strumenti di debugging

Questo passaggio è uno dei pochi punti obbligatori a prescindere dall'utilizzo o meno di Polymorph. XCode offre già un pattern di default per le Safari App Extension, però questo pattern è molto basilare, tant'è che non implementa un sistema di message passing tra lo script interno e quello esterno. Inoltre, seppur venga offerta la scelta di progettare l'applicazione in Objective C o in Swift, il

programmatore non intuisce, in base alle informazioni offerte da XCode, quale dei due linguaggi venga preferito per la progettazione di questo tipo di applicazioni.

Polymorph, seppur non automatizzi totalmente questo passaggio, offre, nel caso l'estensione originale implementi un sistema di scambio dei messaggi tra script interno e esterno, degli script molto più elaborati che sfruttano maggiormente le funzionalità delle Safari App Extension; inoltre gli script prodotti da Polymorph sono in Swift perchè attualmente questo è il linguaggio preferito per la programmazione di applicazioni Apple.

Documentarsi attentamente sul linguaggio di programmazione Swift

Swift è un linguaggio che può cambiare notevolmente la sua sintassi e le sue librerie a seconda della versione utilizzata, trovare documentazione a riguardo non è banale poiché, oltre alla ricerca del significato di certe istruzioni, bisogna anche vedere, nel caso si trovino le istruzioni, a quale versione di Swift si riferiscano. Il codice Swift che usa Polymorph usa la sintassi della quarta versione che, seppur non sia la versione più recente, è una di quelle su cui si trova più facilmente la documentazione.

Decidere come implementare l'interfaccia e gli script (e di conseguenza, quali librerie usare)

Teoricamente, le Chrome Extension e le Safari App Extension sono totalmente prive di punti in comune riguardo il codice che le costituiscono dato che fanno utilizzo di linguaggi estremamente differenti adibiti per scopi differenti: le estensioni di Chrome sono sorrette da linguaggi comuni ai progettisti di siti web, HTML, CSS, Javascript e JSON, tendendo a supportare molto velocemente le loro novità implementative.

Le Safari App Extension, d'altra parte, fanno uso, a livello di interfaccia, il linguaggio XIB, a livello sintattico simile al linguaggio XML, mentre le loro funzionalità sono programmate in Swift (o Objective C), facilitando la loro progettazione ai programmatori di applicazioni Apple, ampiamente abituati con questi due linguaggi.

La mancanza di riciclaggio del codice non è dato solo dai linguaggi differenti, ma anche dalla differente rigorosità: da una parte le estensioni di Chrome usano

linguaggi, ad eccezione del JSON, poco restrittivi consentendo molta libertà al programmatore, invece le Safari App Extension usano linguaggi molto più restrittivi riguardo il rispetto delle regole sintattiche e della dichiarazione di variabili.

Se Javascript ha una tipizzazione estremamente debole, tale da modificare in tempo reale il tipo di dato che possono memorizzare le variabili, d'altro canto Swift ha una tipizzazione molto forte dei dati [92], vietando certi passaggi implementativi applicabili con Javascript. Infatti, seppur il programmatore possa decidere di non dichiarare esplicitamente il tipo di dato delle variabili, sarà Swift a determinare automaticamente il loro tipo in base al valore assegnato e vanta una politica estremamente rigida riguardo il casting di tipo.

Questa politica di casting si esprime tramite `as`, `as!`, `as?` e `is`, tuttavia il primo termine porta molto spesso all'interruzione improvvisa dell'applicazione, quindi è preferibile usare gli altri termini, estremamente differenti tra loro e che possono cambiare notevolmente l'esecuzione del programma: `as!` forza il casting di tipo, portando a un crash dell'applicazione se esso non avverrà con successo.

Il termine `as?` è un casting condizionale, assegnando il tipo richiesto se il casting avviene con successo, altrimenti l'istruzione darà un valore `nil`. L'ultimo termine, `is`, è un termine usato per le istruzioni condizionali. Il typecasting avverrà con successo se e solo se è un downcasting, ovvero passa da una classe più generica a una sua sottoclasse, fare un casting assegnando un tipo "padre" o una classe appartenente al medesimo livello di gerarchia darà un errore. Questo è una delle causi più ricorrenti nei crash di applicazioni Swift e il programmatore troverà pochissime informazioni a riguardo nella documentazione ufficiale [93].

Tuttavia, a discapito di un tempo di avvio più lungo dell'estensione, si può usare uno stratagemma che consenta il riutilizzo di buona parte del codice proveniente da un estensione di Chrome: le Safari App Extension possono agire in due modi differenti, grazie alle opzioni offerte da Info.plist, se si preme il loro bottone, "Command", che esegue istruzioni senza l'utilizzo di un'interfaccia, e "Popover", azione che genera un popup che ospiterà l'interfaccia dell'estensione.

Questo popup, al di là dell'aspetto differente, ha le medesime funzionalità di una qualsiasi applicazione programmabile tramite XCode, quindi può ospitare qualsiasi

libreria e/o framework desideri implementare il programmatore, incluso “WKWebView”, una classe presente nel framework WebKit adibita per il caricamento, sia offline che online, dei file HTML e dei loro allegati, inclusi CSS, Javascript e file riguardanti il font.

Di conseguenza, se il programmatore decidesse di implementare “WKWebView” all’interno del Popover dell’estensione, può senza alcun problema usare i file del progetto originale offrendo la medesima interfaccia dell’estensione originale; tuttavia questo passaggio implementativo non è presente nella documentazione ufficiale e quel poco che il programmatore può trovare a riguardo è difficoltoso da reperire.

Polymorph, quando converte un estensione Chrome adattandola a Safari, implementa tutti i passaggi necessari per utilizzare il file HTML come file di interfaccia e, nelle istruzioni, comunica al programmatore quali librerie dovranno essere aggiunte nel progetto per fare in modo che la compilazione avvenga con successo, che saranno le seguenti: Cocoa, Safari Services, WebKit e AppKit saranno le librerie da aggiungere manualmente.

Oltre a ciò, il programmatore comprenderà assai più facilmente, tramite i pattern offerti da Polymorph, le regole di Swift riguardo l’integrazione delle librerie e la dichiarazione e utilizzo delle loro classi all’interno di un progetto XCode. Tutto ciò che seguirà da questo punto in poi presuppone che il programmatore, per implementare una Safari App Extension partendo da una estensione di Chrome, faccia utilizzo della classe “WKWebView” e di tutte le funzionalità che può offrire questo framework grafico.

Distinguere i frammenti di codice che fanno utilizzo di istruzioni native di Chrome

Le uniche istruzioni Javascript di una Chrome Extension incompatibili con Safari sono le istruzioni native di Chrome; esse si distinguono molto facilmente da due fattori:

- Solitamente hanno la sintassi “chrome._istruzione_”.
- Di solito il loro funzionamento fa utilizzo di callback, rendendole facili da individuare pure osservando i blocchi di programmazione delimitati dalle

loro parentesi graffe.

Se da una parte, quindi, sono estremamente facili da individuare (è sufficiente scrivere “chrome” all’interno della funzionalità di ricerca spesso offerta anche dai blocchi note) d’altro canto trovare tutte le istanze in cui vengono utilizzate queste istruzioni native può non essere banale perché estendono notevolmente le funzionalità degli script Javascript consentendo un ottima sinergia tra di essi e Chrome. Polymorph individua molto facilmente le istruzioni native grazie a un dizionario, col quale le individua e, in base alla loro sintassi, decidere automaticamente quali azioni compiere, mantenendo il codice sostitutivo quanto più fedele possibile a livello semantico al codice originale.

Inoltre, bisogna anche saper distinguere le istruzioni native che si legano fortemente col browser come lo scambio dei messaggi o query riguardanti i tab del browser; queste ultime è molto probabile che necessitino di essere scritte in parte anche tramite Swift perché il “WKWebView”, con le poche istruzioni native a disposizione, può solo ed esclusivamente inviare messaggi contenenti dati e risorse.

Distinguere i frammenti di codice che praticano lo scambio di messaggi

Uno dei passaggi più complicati, nel passare da una Chrome Extension a una Safari App Extension, è lo scambio dei messaggi tra l’estensione e il browser, sia a causa di una separazione assai più netta che per il necessario utilizzo di Swift come intermediario tra le due applicazioni, rendendo necessario un doppio message passing, rispettivamente tra il “WKWebView” e l’applicazione che lo ospita e tra quest’ultima e il browser.

Questi frammenti, inoltre, saranno gli unici che fanno utilizzo di istruzioni Javascript native di Safari. Questi frammenti di codice, seppur sempre facilmente individuabili tramite la ricerca del termine “chrome”, devono essere controllate attentamente per vari fattori:

- È probabile che gli invii e le ricezioni dei messaggi abbiano molte callback annidate, rendendo necessaria un attenta analisi di tutti quei casi dove le callback avvengano con successo e in quali casi esse possano fallire a causa dei vari scenari possibili durante l’esecuzione.

- Javascript ha una tipizzazione dei dati molto più debole di Swift, quindi bisogna controllare attentamente anche i dati che vengono scambiati, cercando in Swift un tipo di dato che possa memorizzarli tramite il downcasting.

Il downcasting dei dati è fondamentale perché, senza di esso, Swift non può inviare con successo i messaggi al content script adibito a riceverli; questo è uno dei pochi passaggi su cui Polymorph, nonostante le sue funzionalità, non può completamente sostituire il programmatore poichè ad ogni dato scambiato tramite Javascript si deve trovare un corrispettivo Swift il più fedele possibile al tipo di dato originale; tuttavia, presentando un pattern per il downcasting, aiuterà il programmatore a programmare più facilmente il casting di tipo sui dati ricevuti dallo script all'interno del "WKWebView".

Sostituirli le istruzioni native con codice valido per Safari

Se da una parte l'interfaccia è solo da impostare correttamente tramite il "WKWebView", tutt'altro discorso vale per il codice Javascript. Individuate le istruzioni native e identificate quelle fortemente legate al browser, bisogna convertire tutte queste istruzioni con sintassi interpretabili da Safari.

L'identificazione di istruzioni strettamente legate al browser come lo scambio dei messaggi e differenziare tali frammenti di codice dalle altre istruzioni native può aiutare il programmatore a ridurre al minimo la scrittura di codice Swift, di conseguenza si riducono gli scambi di messaggi tra il "WKWebView" e l'applicazione che lo ospita, avvantaggiando le performance.

Per esempio, le istruzioni relative alla memorizzazioni di dati offerte da Chrome possono essere sostituite con le istruzioni di Web Storage API, supportate da Safari; l'unico momento cruciale in cui praticare lo scambio dei messaggi, in questo caso, è quando viene aggiornata la memoria con gli ultimi dati elaborati in modo che anche l'applicazione ospitante il "WKWebView" conosca i dati aggiornati da poter usare poi sulle pagine visitate tramite il content script.

Quindi, spesso, esistono delle buone alternative alle istruzioni native di Chrome che non solo funzionano sulla maggioranza dei browser, ma aiutano anche ad aumentare la riciclabilità del codice, avvantaggiando il debugging e

l’implementazione di queste estensioni anche su browser con motori estremamente differenti.

Esistono altre istruzioni, invece, che possono essere programmabili solo tramite l’applicazione ospitante il “WKWebView”, come la richiesta di aggiornare un tab del browser. Se un'istruzione simile tramite le estensioni di Chrome richiede poche righe di istruzioni native Javascript, con le Safari App Extension tale funzionalità si può ottenere solo tramite il linguaggio Swift in maniera assai differente.

Se da una parte Chrome offre pure funzionalità tali da prendere i tab desiderati tramite query, con le Safari App Extension è possibile solo memorizzando tutte le pagine visitate tramite un array globale e, quando si hanno i dati aggiornati provenienti dallo script interno, inviare a essi, tramite un ciclo for, le azioni da eseguire; quindi, per dire un esempio, non c’è la possibilità di modificare solo ed esclusivamente il tab attivo.

Polymorph sostituisce le istruzioni native di Chrome con, se possibile, istruzioni ben supportate dalla maggioranza dei browser, in modo da ridurre l’utilizzo dello scambio dei messaggi tra l'applicazione ospitante il “WKWebView” e lo script Javascript all’interno di quest’ultimo e avvantaggiando le possibili future conversioni dell’applicazione su altri browser con standard differenti per la programmazione di estensioni.

Implementare il sistema di doppio scambio dei messaggi

Questo passaggio è estremamente complicato a causa della già citata politica rigida di casting di tipo di Swift. Purtroppo la documentazione poco chiara al riguardo non aiuta il programmatore a comprenderla e di conseguenza ridurre al minimo gli errori. Polymorph offre già, basandosi sull’estensione Joo Janta 200, un sistema di scambio dei messaggi funzionante includendo le istruzioni principali con cui implementarlo.

Tuttavia il programmatore dovrà adattare una delle funzioni scritte in Swift, pensata per ricevere i messaggi dal “WKWebView” per poi inviarli allo script esterno, in base ai tipi differenti di dato che il programmatore vorrebbe venissero inviati; questo frammento di codice da adattare riguarda esclusivamente il casting di tipo.

Passare da una Safari App Extension a una Chrome Extension

Per convertire una Safari App Extension rendendola una Chrome Extension prima di tutto i requisiti sono assai più accessibili poiché bastano solo:

- Un browser basato su Chromium
- Per modificare i file HTML e Javascript il programmatore può anche usare un semplice blocco note, seppur siano caldamente consigliati strumenti più opportuni.

Riguardo il progetto originale, bisogna distinguere i due metodi di progettazione principali con cui può essere realizzata una Safari App Extension.

L'interfaccia è XIB e ha gli script totalmente scritti in Swift

Questo caso, non ricoperto da Polymorph, viene comunque citato poiché questo tipo di progettazione è quello "ufficiale", quello indicato da Apple per sfruttare appieno il loro modello di estensione. Se l'estensione è progettata in questo modo, ciò implica la totale riscrittura del codice, sia per il markup che per lo script, del codice originale possono essere preservate solo le idee implementative, seppur con una sintassi assai differente; al di là delle idee, tuttavia, la progettazione da zero di una nuova estensione è inevitabile.

L'estensione è progettata in modo da essere simile a una estensione di Chrome

Se l'estensione sfrutta i trucchi delle App Extension presentando un'interfaccia HTML e script Javascript grazie all'utilizzo del framework "WKWebView", a questo punto la conversione dal progetto originale a un estensione per Chrome mantenendo inalterati vari frammenti di codice è possibile, seppur richieda attenzione riguardo lo scambio dei messaggi.

Su Chrome, infatti, non sarà più necessario fare il doppio scambio dei messaggi poiché non presenta una separazione così netta tra le estensioni e il browser. Inoltre, se si fa utilizzo di un particolare passaggio implementativo relativo alla memorizzazione dei dati, lo scambio di messaggi può addirittura essere totalmente eliminato, rendendo il debugging molto meno complicato. I passaggi sono i seguenti:

Creare il file manifest.json, includendo i link ai file di markup e script

Come visto precedentemente, l'unico file obbligatorio in una estensione di Chrome è il manifest.json, colui che memorizza tutte le autorizzazioni e i permessi concessi all'estensione. Tuttavia, oltre a vantare una sintassi rigorosa, essa è piuttosto differente dal file Info.plist. Quest'ultimo file spesso, tra l'altro, grazie a XCode ha il processo di modifica semplificato perchè anzichè modificare il codice vero e proprio il programmatore, tramite un'interfaccia intuitiva, può decidere di aggiungere velocemente strutture e dati che facciano riferimento ai requisiti richiesti in maniera intuitiva, a differenza del manifest.json spesso scritto manualmente.

Polymorph, osservando la sintassi dell'Info.plist, convertirà i permessi più importanti in permessi JSON mantenendo la medesima semantica, velocizzando la procedura di creazione e debugging dei permessi. Solitamente, se si usa un file HTML come interfaccia, una buona soluzione è quella di mantenere i suoi file in una cartella,in modo da rendere più chiara la struttura del progetto. Polymorph saprà copiare e incollare quella cartella sul percorso di output impostato dall'utente e adattare i link del manifest.json relativi a file di progetto.

Analizzare la semantica delle istruzioni scritte in Swift e delle istruzioni native Javascript di Safari

Le istruzioni Javascript native di Safari sono pochissime, appena un paio, dedicate esclusivamente alla funzionalità di scambio dei messaggi e si identificano facilmente perché contengono la stringa "safari". Tutt'altro discorso vale per le istruzioni Swift, di cui bisognerà comprendere la loro semantica e individuare esclusivamente:

- Le istruzioni relative all'interazione tra l'estensione e il browser come lo scambio dei messaggi o la memorizzazione dei dati.
- L'istruzione che carica il file HTML adibito come interfaccia di default.

Tutte le altre istruzioni, pensate ad esempio per inizializzare il "WKWebView", non servono per la progettazione di una Chrome Extension, essendo HTML e Javascript i suoi linguaggi principali, basta saper collegare correttamente questi file al manifest.json.

Polymorph individua facilmente l'istruzione che carica il file HTML, tuttavia, a causa della sintassi elaborata di Swift, riguardo le istruzioni per lo scambio dei messaggi analizza solo ed esclusivamente le due istruzioni native Javascript che hanno un comportamento molto semplificato, quindi il progettista dovrà osservare attentamente se l'intermediario Swift compia azioni particolarmente elaborate sui messaggi.

Sostituire queste istruzioni con altre compatibili con Chrome

Arrivati a questo punto, in teoria, si dovrebbero modificare solo ed esclusivamente i frammenti di codice relativi allo scambio dei messaggi, tuttavia, per quello che si è detto prima, la sintassi Swift è molto difficile da interpretare con un programma, quindi Polymorph non saprà sostituire le istruzioni Swift con istruzioni Javascript tali da mantenere la medesima semantica, questo anche perchè pure il sistema di scambio dei messaggi di Chrome è estremamente elaborato.

Infatti Polymorph segue una strada alternativa: avendo aggiunto nel manifest.json l'autorizzazione "Storage", l'applicazione può sfruttare lo storage locale dell'estensione come intermediario tra lo script interno e il content script. Questa alternativa ha dei pro e dei contro: da una parte il sistema di scambio di dati e messaggi verrà notevolmente semplificato, quasi annullato: infatti le uniche informazioni ora necessarie da inviare tramite messaggi consistono nell'avvisare il browser di aggiornare la pagina in modo da mettere in azione il content script o, sempre aggiornando la pagina, annullare le modifiche che applicò quest'ultimo.

Tuttavia ciò può avvenire con successo se e solo se viene utilizzato lo storage locale di Chrome, con la memoria locale dell'API Web Storage ciò non è possibile, quindi il programmatore dovrà sostituire tutte le istruzioni di memoria locale adattandole all'API di Chrome.

Polymorph non ha una simile potenza di lettura, quindi manterrà le istruzioni originali. Tuttavia, nel caso trovi un istruzione che aggiorna la memoria locale, le allega un frammento di codice con istruzioni Javascript native di Chrome che fa uso della sua memoria locale e di una query dei tab. Questo frammento di codice autorizza il browser ad aggiornare la pagina attiva in modo da annullare o applicare le modifiche del content script tramite i dati aggiornati.

Capitolo 6
Valutazione di Polymorph: un caso studio

Polymorph semplifica questa operazione: infatti, oltre a convertire le autorizzazioni dell'estensione, scritte in maniera differente a seconda del modello di progettazione considerato, adattandole al modello desiderato, traduce pure le istruzioni native Javascript, dove per "native" si intendono le istruzioni valide solo per alcuni browser; alcune istruzioni Javascript sono valide solo per Safari, altre valide solo per Chrome e, di conseguenza, i browser compatibili con le sue estensioni; Polymorph, traducendo queste istruzioni native, le può sostituire con altre istruzioni native compatibili col browser desiderato (infatti, spesso queste istruzioni native sono necessarie per rendere funzionante l'estensione), oppure in istruzioni supportate da tutti i browser.

Polymorph è un software che analizza le estensioni per i browser e, in base alle impostazioni dell'utente, applica delle modifiche a livello sintattico riguardo le istruzioni Javascript mantenendo la maggiore fedeltà possibile alla semantica originale e aggiungere i file necessari per la costruzione di un estensione per il browser desiderato: per esempio, scegliendo una Safari App Extension da convertire per Chrome, viene aggiunto un manifest.json dato che Chrome non è compatibile col file Info.plist.

Polymorph è basato principalmente su tre file Javascript: index.js, contenente l'interfaccia e i controlli sugli input dell'utente; polymorph1.js, il fulcro del funzionamento dell'applicazione che, dopo aver copiato i file inserendoli nella cartella di output definita dall'utente, fa tutte le modifiche opportune, e polymorph2.js, che assiste polymorph1.js per la creazione di file che richiedono stringhe multilinea e nel trovare le istruzioni native da modificare. D'ora in poi si elencano le autorizzazioni convertibili, le istruzioni native considerate da Polymoprh e il risultato che stampa l'applicazione a seconda dello standard tecnico scelto dall'utente.

Come riferimento per la conversione di estensioni è stata considerata l'estensione Joo Janta 200, disponibile per Chrome e Firefox: lo scopo di questa estensione è di trovare, nelle pagine web visitate dall'utente, parole corrispondenti a quelle presenti nella sua tabella; se avviene una corrispondenza, l'estensione cambia quella parola con un'altra, sempre presente nella sua tabella.

Essendo un'estensione basata sia sullo scambio di messaggi tra il suo script interno e il suo script esterno sia sulla modifica in tempo reale delle pagine web, Joo Janta 200 è piuttosto elaborata sfruttando tutti i paradigmi ormai consolidati delle estensioni per i browser.

Joo Janta, a livello di architettura interna, è composto da un file HTML con la funzione di interfaccia, da uno script interno e uno script esterno; l'interfaccia è composta di vari tab in modo da rendere l'estensione intuitiva anche per gli utenti che si approcciano per la prima volta all'utilizzo dell'estensione: infatti il primo tab permette di attivare o no Joo Janta, il secondo permette di modificare la tabella delle stringhe, il terzo contiene i link di Google Sheet con cui caricare tabelle di stringhe predefinite oppure resettare le impostazioni dell'estensione, il penultimo presenta il changelog, la licenza e le fonti delle librerie usate dall'estensione, infine l'ultimo tab elenca le istruzioni per far funzionare Joo Janta. Un ulteriore tab, visualizzabile solo durante il debugging dell'estensione, può stampare messaggi utili per lo sviluppatore.

Lo script interno memorizza lo stato attivo-inattivo dell'estensione, le parole chiave da cercare affiancate da quelle sostitutive e i link di Google Sheet a cui Joo Janta può riferirsi per caricare ulteriori tabelle di stringhe. Inoltre lo script

inizializza l’interfaccia con i dati di default, se Joo Janta viene usato per la prima volta, oppure con i dati che inserì l’utente l’ultima volta che usò l’estensione.

Per le tabelle lo script fa uso della libreria JsGrid che fornisce alcune istruzioni utili per la loro gestione come la modifica di righe e colonne o l’inserimento di bottoni che facilitano la modifica del contenuto dentro le caselle oppure eliminare le righe. Quando si attiva l’estensione, lo script interno avvisa il browser di ricaricare la pagina attualmente visibile dall’utente.

Durante la ricarica di questa pagina, lo script esterno, che condivide la medesima memoria dello script interno e nota lo stato attivo di Joo Janta, trova all’interno della pagina stringhe coincidenti con quelle della tabella e, nel caso ciò avvenga, le sostituisce con le opportune stringhe sostitutive. Quando l’estensione viene disattivata, la pagina attualmente visibile dall’utente viene ricaricata e lo script esterno, notando lo stato disattivo dell’estensione, non agisce sulla pagina mantenendola inalterata.

Polymorph quando converte un estensione pensa principalmente alle autorizzazioni e agli script Javascript. A causa dell’enorme diversità tra i due modelli, le autorizzazioni sono anch’esse differenti, in particolare quelle riguardanti il legame tra esse e il browser come la possibilità di accedere al tab attivo. Questo è il manifest.json dell’estensione originale:

```
{
   "browser_action": {
      "default_icon": {
         "64": "images/glasses-p-64.png"
      },
      "default_popup": "joojanta.html",
      "default_title": "The Joo Janta 200 Super-Chromatic Nuisance Sensitive Sunglas
   },
   "content_scripts": [ {
      "all_frames": true,
      "css": [ "css/content.css" ],
      "exclude_globs": [ "*.google.*" ],
      "js": [ "js/content.js" ],
      "matches": [ "\u003Call_urls>" ],
      "run_at": "document_idle"
   } ],
   "description": "The Joo Janta 200 Super-Chromatic Nuisance Sensitive Sunglasses, (
   "icons": {
      "64": "images/glasses-p-64.png"
   },
   "key": "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAiSQ0QxdoOhcTLl3Slyqk8BVE2ebPE(
   "manifest_version": 2,
   "name": "Joo Janta 200",
   "permissions": [ "activeTab", "storage" ],
   "short_name": "Joo Janta 200",
   "update_url": "https://clients2.google.com/service/update2/crx",
   "version": "1.2"
}
```

Le seguenti immagini, invece, raccolgono un frammento del file Info.plist creato da Polymorph in base ai dati del manifest mostrato precedentemente, raccolti nel dizionario identificato dalla chiave “NSExtension”.

```
<key>SFSafariContentScript</key>
<array>
  <dict>
    <key>Script</key>
    <string>extension/js/content.js</string>
    <key>Excluded URL Patterns</key>
    <array>
      <string>http://*.google.*</string>
      <string>https://*.google.*</string>
    </array>
  </dict>
</array>
<key>SFSafariStyleSheet</key>
<array>
  <dict>
    <key>Style Sheet</key>
    <string>extension/css/content.css</string>
    <key>Excluded URL Patterns</key>
    <array>
      <string>http://*.google.*</string>
      <string>https://*.google.*</string>
    </array>
  </dict>
</array>
<key>SFSafariToolbarItem</key>
<dict>
  <key>Action</key>
  <string>Popover</string>
  <key>Identifier</key>
  <string>Button</string>
  <key>Image</key>
  <string>extension/images/glasses-p-64.png</string>
  <key>Label</key>
  <string>Joo Janta 200</string>
</dict>
<key>SFSafariWebsiteAccess</key>
<dict>
  <key>Level</key>
  <string>All</string>
</dict>
```

Come si nota, la struttura dei file è molto differente. L'elemento che si nota maggiormente è la dichiarazione dello schema degli indirizzi URL, necessaria per compilare con successo l'applicazione. Polymorph adatta le autorizzazioni strettamente necessarie come gli URL su cui far agire l'estensione, dichiarare il percorso per l'immagine che avrà il ruolo di icona (percorso leggermente modificato tramite l'aggiunta della cartella "extension").

Riguardo l'implementazione di un file HTML che fa veci di interfaccia ciò è possibile solo tramite il seguente frammento di codice in Swift aggiunto da Polymorph:

```
    var myWebView: WKWebView = WKWebView()

    static let shared: SafariExtensionViewController = {
        let shared = SafariExtensionViewController()
        shared.preferredContentSize = NSSize(width:310, height:350)
        return shared
    }()
    override func loadView() {
        var myWebView1: WKWebView
        let webConfiguration = WKWebViewConfiguration()
        let userContentController = WKUserContentController()
        userContentController.add(self, name: "script")
        webConfiguration.userContentController = userContentController

        myWebView1 = WKWebView(frame: .zero, configuration: webConfiguration)
        myWebView = myWebView1
        myWebView.uiDelegate = self
        view = myWebView
        myWebView.navigationDelegate = self
        myWebView.frame = view.frame
    }
    override func viewDidLoad() {
        super.viewDidLoad()
        let myURL = Bundle.main.url(forResource: "joojanta", withExtension: "html", subdirectory: "extension")!

        myWebView.loadFileURL(myURL, allowingReadAccessTo: myURL.deletingLastPathComponent())
        NSLog("la variabile mywebview è ((String(describing: myWebView)))")



    }
```

Tuttavia la conversione di Polymorph bada solo agli elementi strettamente necessari per il funzionamento dell'estensione. I campi descrittivi all'interno del manifest come "description" attualmente non vengono considerati, seppur le loro informazioni permettano un maggior approfondimento del funzionamento dell'applicazione.

I vantaggi e gli svantaggi di questa conversione si presentano anche passando da una Safari App Extension a una Chrome Extension: considerando il frammento Info.plist mostrato precedentemente e convertendolo il risultato è il seguente:

```
{
  "name": "Joo Janta 200",
  "version": "1.2.1",
  "browser_action": {
    "default_popup": "extension/joojanta.html",
    "default_icon": {
      "64": "extension/images/glasses-p-64.png"
    }
  },
  "icons": {
    "64": "extension/images/glasses-p-64.png"
  },
  "content_scripts": [
    {
      "matches": [
        "<all_urls>"
      ],
      "exclude_globs": [
        "http://*.google.*",
        "https://*.google.*"
      ],
      "all_frames": true,
      "run_at": "document_idle",
      "js": [
        "extension/js/content.js"
      ]
    }
  ],
  "permissions": [
    "activeTab",
    "storage"
  ],
  "manifest_version": 2
}
```

La differenza sostanziale del Joo Janta 200 “riconvertito” rispetto all’estensione originale, considerando gli elementi essenziali, consistono nella dichiarazione dello schema negli URL, ereditata dalla estensione per Safari; Polymorph tiene conto anche della necessità di dichiarare il percorso dell’icona su un paio di campi differenti, infatti “default_icon” è pensato per visualizzare tale immagine come icona per identificare facilmente l’estensione posta vicino alla barra degli indirizzi, mentre “icons” permette di riutilizzare l’immagine quando si apre il menù di

gestione delle estensioni. Anche in questo caso Polymorph considera solo gli elementi strettamente necessari, tralasciando quelli descrittivi.

Polymorph, quando adatta gli script di Joo Janta 200 al browser desiderato, si preoccupa principalmente dello scambio di messaggi e della memorizzazione delle parole chiave, estremamente differente tra Chrome e Safari dato che quest'ultimo rende necessaria la conoscenza del linguaggio di programmazione Swift, di cui è poco documentato il suo contributo nelle Safari App Extension.

Riguardo la memorizzazione delle parole chiave di Joo Janta, Polymorph verifica la presenza di istruzioni native come `chrome.local.storage.get`, `chrome.local.storage.set`, `browser.local.storage.get` e `browser.local.storage.set`, in modo da scriverle seconda la sintassi dell'API Web Storage, facendo uso quindi di `localStorage.getItem` e `localStorage.setItem`.

Di seguito verrà riportato un esempio semplice di tale modifica, l'istruzione originale qua sotto è pensata per funzionare su Chrome:

```
chrome.storage.local.get('joojanta', function(data) {//codice});
```

Questa istruzione permette di prendere dalla memoria locale dell'estensione ciò che è identificato dalla chiave "joojanta" e di inserirlo all'interno di "data", argomento della callback identificata da una funzione. Questa istruzione non funziona su Safari rendendo necessaria la sua riscrittura cercando di mantenere la sua semantica. La soluzione più vicina a questa, nel caso si programmi una Safari App Extension, è la seguente:

```
var data = JSON.parse(localStorage.getItem('joojanta'));
//codice
```

Questa nuova sintassi non altera la semantica e non richiede nessuna modifica al codice originariamente eseguito dalla callback, l'unica modifica riguarda l'utilizzo di una memoria locale differente: anziché usare quella di Chrome, viene usata quella dell'API Web Storage, pienamente supportata da Safari.

Polymorph, quando usa l'API Web Storage, si preoccupa di memorizzare tutti i dati in una struttura JSON in modo da uniformare la memorizzazione, per questo è

necessario l'utilizzo di `JSON.parse`, istruzione che trasforma la struttura JSON in un oggetto contenente dati di tipo differente.

Non sempre però questa modifica sarà scontata, infatti il seguente frammento di codice, compresso in alcune porzioni non inerenti all'argomento su cui discutere, è colui che inizializza l'interfaccia di Joo Janta:

```
39  chrome.storage.local.get('joojanta', function(data) {
40    if (!data || !data.joojanta) {
41      log("Could not find stored preferences. Resetting to defaults")
42      var data = {
43        joojanta: {
44          replacements: baseReplacements,
45          sets: baseSets,
46          active : false
47        }
48      } ;
49      saveData(baseReplacements, baseSets, false) ;
50    }
51
52    log("We're going to use these data: ", data) ;
53    $('#slider')[0].checked = data.joojanta.active;
54
55 ›  $("#repGrid").jsGrid({…});
75
76 ›  $("#setGrid").jsGrid({…});
102   log("Finished preparing for display") ;
103 });
```

Questa istruzione in Joo Janta non è contenuta in nessun blocco (ovvero, non è contenuta dentro delle parentesi graffe) e viene eseguita immediatamente; non può essere sostituita con un semplice `localStorage.getItem` perché bisogna mantenere la struttura del programma, quindi, oltre alla memoria locale, bisogna pensare a un istruzione che possa simulare la sua esecuzione immediata senza alterare eccessivamente i blocchi identificati dalle parentesi graffe. Una buona sostituzione data da Polymorph, nel caso noti che una di queste istruzioni non appartenga a nessun blocco di programmazione, è questa:

```
(function() {
var data = JSON.parse(localStorage.getItem('joojanta'));
  if (!data || !data.joojanta) {
    log("Could not find stored preferences. Resetting to defaults")
    var data = {
      joojanta: {
        replacements: baseReplacements,
        sets: baseSets,
        active : false
      }
    } ;
    saveData(baseReplacements, baseSets, false) ;
  }

  log("We're going to use these data: ", data) ;
  $('#slider')[0].checked = data.joojanta.active;

  $("#repGrid").jsGrid({…});

  $("#setGrid").jsGrid({…});
  log("Finished preparing for display") ;
})();
```

Per modificare questi frammenti di codice per adattarle a Safari, Polymorph si preoccupa sintassi dei blocchi di programmazione, infatti, se `chrome.local.storage.get` non è contenuta in nessun blocco di programmazione, la si modifica con `function()` mantenendo la sua esecuzione immediata, ma hanno una sintassi differente delle parentesi: la sintassi dell'istruzione nativa di Chrome presente qua sotto.

```
chrome.storage.local.get('joojanta', function(data) {
 …
});
```

Presenta una sintassi diversa dalla seguente:

```
(function() {
 var data = JSON.parse(localStorage.getItem('joojanta'));
 …
})();
```

Riguardo il passaggio da Safari a Chrome il tutto è più semplice perchè Chrome supporta ampiamente l'API Web Storage non richiedendo nessuna modifica sostanziale, ma analizzare il sistema di scambio dei messaggi di una Safari App Extension è molto complicato sia a causa del doppio scambio di messaggi che per la complessità di analisi sintattica delle istruzioni Swift che fanno da intermediario per lo scambio dei dati; in questo caso Polymorph può eliminare lo scambio dei messaggi mettendo in condivisione lo storage tra lo script interno e il content script facendo utilizzo delle istruzioni native di Chrome.

Tuttavia, sostituire tutte le istruzioni che fanno uso del Web Storage non è banale, in particolare per quelle riguardanti la raccolta dei dati. In questo caso la difficoltà è dettata da un particolare fattore: seppur le istruzioni Web Storage non usino callback, spesso sono all'interno di altre istruzioni spesso dedite a un casting di tipo, come visto poco prima tramite l'utilizzo di `JSON.parse`; ciò implica la necessità di separare l'istruzione dello storage locale dal resto portando a due istruzioni consecutive, Polymorph non ha una capacità di lettura e riconoscimento delle istruzioni così sofisticato.

Quindi, per scelta implementativa, quando Polymorph incontra un'istruzione che faccia uso di Web Storage, l'istruzione non verrà alterata; infatti per implementare lo storage condiviso tramite istruzioni native di Chrome Polymorph analizza la presenza di un'altra istruzione, presente nativamente nelle Safari App Extension, spesso usata subito dopo la memorizzazione dei dati elaborati che è la seguente, su cui verrà fatto un approfondimento successivamente.

```
window.webkit.messageHandlers.script.postMessage(base);
```

Riguardo lo scambio dei messaggi, Joo Janta non fa un vero e proprio message passing, ma permette sia allo script interno che a quello esterno di prendere la tabella di stringhe dalla medesima memoria locale, questo grazie alla dichiarazione affermativa dell'autorizzazione "Storage" nel manifest.json; infatti, quando l'utente attiva o disattiva l'estensione, Joo Janta esegue le istruzioni riportate sopra aggiornando il content script sullo stato dell'estensione, uno dei dati memorizzati nella memoria locale.

Joo Janta, dopo aver salvato i dati, che includono lo stato attivo o inattivo dell'estensione, avvisa Chrome, tramite una callback, di ricaricare la pagina

attualmente visitata dall'utente. Infatti, ogni volta che l'utente visita tramite il browser una pagina, lo script esterno esegue immediatamente questa istruzione, non essendo contenuta in nessun blocco di programmazione.

```
var allPatterns ;
chrome.storage.local.get('joojanta', function(data) {
  if (data.joojanta && data.joojanta.active) {
    log('Now active') ;
    allPatterns = prepare(data.joojanta.replacements) ;
    replaceAll(allPatterns) ;
  } else {
    log('Not active. Activate it by clicking on the purple sunglasses on your browser bar.') ;
  }
}) ;
```

Ciò che fa questa istruzione è molto semplice: se l'istruzione ha raccolto dati dalla memoria locale e vede che l'utente ha deciso di attivare le funzionalità di Joo Janta, comincia a cercare le parole chiave sostituendole con stringhe desiderate dall'utente, altrimenti mantiene intatta la pagina appena visitata.

In Safari la procedura, in particolare se si desidera mantenere l'interfaccia HTML, è molto più complicata, infatti la memoria locale dell'estensione è nettamente separata da quella del browser richiedendo l'implementazione di due scambi di messaggi: quello da Joo Janta al codice Swift, con messaggi a senso unico, e quello tra Swift e lo script esterno, dove entrambi possono inviare e ricevere messaggi.

Il primo scambio di messaggi, quello tra lo script interno dell'estensione e l'estensione vera e propria, è molto semplice da implementare, dato che l'estensione non può far altro che inviare messaggi coi dati a sua disposizione, mentre Swift ricoprirà sempre il ruolo del destinatario ricevendo i messaggi. Lo script interno dell'estensione, per inviare i messaggi, usa una semplice istruzione:

```
window.webkit.messageHandlers.script.postMessage(base);
```

Questa Istruzione ha due argomenti: il primo è l'identificativo "script", incluso tra "messageHandlers" e "postMessage", che aiuta Safari a identificare il messaggio, fare le opportune verifiche e inviare allo script esterno; l'argomento "base", invece, contiene i dati dell'estensione, quelli che il progettista desidera inviare: il suo stato, le parole chiave correlate con quelle sostitutive e i link Google Sheet.

Polymorph, passando da Chrome a Safari, piazzerà questa istruzione quando convertirà un `chrome.storage.local.set`; in tal caso, oltre a convertire quest'ultima istruzione con una offerta dall'API Web Storage, aggiunge l'invio del messaggio informando l'applicazione contenente l'interfaccia HTML dei dati elaborati. Quindi il seguente frammento:

```
248       if (r)
249         base.joojanta.replacements = r ;
250       if (s)
251         base.joojanta.sets = s ;
252       if (a!==undefined && a!==null)
253         base.joojanta.active = a ;
254       chrome.storage.local.set(base, f)  ;
255     });
```

Diviene grazie a Polymorph:

```
249       if (r)
250         base.joojanta.replacements = r ;
251       if (s)
252         base.joojanta.sets = s ;
253       if (a!==undefined && a!==null)
254         base.joojanta.active = a ;
255   localStorage.setItem('joojanta', JSON.stringify(base));
256   window.webkit.messageHandlers.script.postMessage(base);
```

Il contributo di `JSON.parse` e `JSON.stringify` è molto importante: infatti memorizzare i dati come una stringa JSON facilita la gestione della memoria locale, invece trasformare in dati ciò che si riceve dalla memoria locale facilita la loro modifica.

La gestione di questi messaggi tramite Safari è necessario gestirla con uno script Swift, SafariExtensionViewController.swift. Il seguente frammento di codice, oltre a ricevere il messaggio, deve eseguire un casting su tutti i dati ricevuti dal messaggio; questo passaggio è obbligatorio per rendere l'invio di un nuovo messaggio allo script esterno di Joo Janta privo di bug. Di seguito c'è tutto il codice che gestisce questo passaggio che Polymorph fornisce di default, funzionante con Joo Janta 200:

```
68      func userContentController(_ userContentController:
        WKUserContentController, didReceive message: WKScriptMessage) {
69
70
71          if message.name == "script"{
72              NSLog("condition: (body)")
73              body = message.body as! [String : Any]
74              let joojanta = body["joojanta"] as! [String : Any]
75              let active = joojanta["active"] as! Int
76              for p in pages{
77                  if (active == 1) {
78                      p.dispatchMessageToScript(withName: "Joojanta", userInfo:
                        (body ))
79                  }
80                  else{
81                      p.reload();
82                  }
83                  NSLog("INVIATO SCRIPT A PAGINA (p)")
84              }
85          }
86
87
88      }
```

Il comportamento è identico a quello dell'estensione originale: se lo stato dell'estensione è attivo, lo script esterno, appena riceve il messaggio, ricarica la pagina applicando le modifiche; altrimenti Safari ricarica la pagina attualmente visitata dall'utente.

Inoltre, in Safari, è necessario tenere in un array le pagine che visiterà l'utente durante la sessione perché, a differenza di Chrome, Safari non ha istruzioni con cui eseguire delle query sui suoi tab; questo array, identificato da "pages" nel frammento di codice esposto prima, viene inizializzato da Safari quando l'utente, aprendo una sessione col browser, visualizza l'interfaccia di Joo Janta. La variabile "pages" proviene da un altro file Swift, SafariExtensionHandler.swift, ed è inizializzato tramite questa dichiarazione:

```
var pages: [SFSafariPage] = []
```

Il secondo scambio di messaggi si conclude quando la funzione in Swift, concluso il casting di tutti i dati contenuti nel messaggio, può inviare questi ultimi allo script esterno che, se inizialmente è scritto in questo modo:

```
var allPatterns ;
chrome.storage.local.get('joojanta', function(data) {
  if (data.joojanta && data.joojanta.active) {
    log('Now active') ;
    allPatterns = prepare(data.joojanta.replacements) ;
    replaceAll(allPatterns) ;
  } else {
    log('Not active. Activate it by clicking on the purple sunglasses on your browser bar.') ;
  }
}) ;
```

Ora, tramite Polymorph, ha questa sintassi:

```
var allPatterns ;
document.addEventListener('DOMContentLoaded', function(event) {
safari.extension.dispatchMessage('newPage');
safari.self.addEventListener('message', function(event){
  if (event.message.joojanta && event.message.joojanta.active) {
    log('Now active') ;
    allPatterns = prepare(event.message.joojanta.replacements) ;
    replaceAll(allPatterns) ;
  } else {
    log('Not active. Activate it by clicking on the purple sunglasses on your
    browser bar.') ;
  }
});
}) ;
```

Il `document.addEventListener` è un istruzione necessaria per lo script esterno e le sue funzionalità sulla pagina visitata: senza questa istruzione le modifiche non vengono attuate. Al suo interno, oltre ad avvisare l'estensione di aver visitato una pagina nuova (che può essere inclusa nell'array delle pagine se risulta differente dalle pagine visitate precedentemente) può ricevere i messaggi tramite l'istruzione `safari.self.addEventListener` e far agire lo script a seconda dello stato dell'estensione: se è disattivata, non fa niente, altrimenti applica le modifiche.

Come si nota, non viene modificata solo l'istruzione nativa, ma anche i dati coinvolti, questo a causa dell'identificativo differente: quindi `data.joojanta` e `data.joojanta.active` diventano `event.message.joojanta` e `event.message.joojanta.active`.

Da notare l'aggiunta di un'ulteriore chiusura di parentesi a causa dell'apertura di un nuovo blocco di programmazione, la ricezione dei messaggi è dentro due blocchi di programmazione, non in uno solo. La ricerca del primo blocco di programmazione effettivamente eseguito funziona in maniera similare a ciò che venne esposto per il background script.

Per trovare la chiusura di un blocco aperto da un'istruzione, Polymorph utilizza la funzione indexBrackets; questa funzione localizza la chiusura del blocco di programmazione facilitando la correzione, l'eliminazione o l'aggiunta di chiusure. Invece il blocco di istruzioni contenuto nel `chrome.tabs.query` visualizzato qui sotto:

```
132  $('#slider').on('change', function(event) {
133    saveData(null, null, this.checked, function() {
134    chrome.tabs.query(
135      { active: true,  currentWindow: true },
136      function(tabs) {
137        chrome.tabs.executeScript(
138          tabs[0].id,
139          { code: 'document.location.reload() ;' }
140        );
141      })
142  //    setTimeout("window.close()", 500) ;
143    })
144  }) ;
```

In Safari è totalmente inutile in quanto questa procedura può essere compiuta solo tramite la programmazione Swift, quindi verrà totalmente omesso. Questa soluzione tuttavia è troppo semplicistica e non riconosce frammenti di codice tali da rendere più personalizzato il comportamento dell'estensione come, ad esempio, l'apertura in una nuova finestra della pagina attiva.

Passando da Safari a Chrome, lo scambio dei messaggi è notevolmente semplificato poiché non sono presenti i problemi riguardanti il doppio scambio dei

messaggi causato da Swift; Polymorph, nel convertire il Joo Janta per Safari a Chrome, cercherà di semplificare il più possibile questo passaggio a vantaggio del progettista.

Proprio la versione originale di Joo Janta propone uno scambio dei messaggi molto semplice e al tempo stesso efficiente quindi Polymorph si ispirerà alle sue soluzioni per convertire lo scambio dei messaggi programmato per le estensioni di Safari. Inoltre, altro elemento importante, convertire un estensione Safari per Chrome è più semplice per le pochissime istruzioni native Javascript di Safari, quindi molte istruzioni come `localStorage.getItem` non richiedono alcuna modifica. Una delle istruzioni che richiede modifiche sostanziali è la seguente:

```
255  localStorage.setItem('joojanta', JSON.stringify(base));
256  window.webkit.messageHandlers.script.postMessage(base);
257
```

Per sostituirla è necessario fare una query sui tab di Chrome nel seguente modo:

```
257  localStorage.setItem('joojanta', JSON.stringify(base));
258  chrome.storage.local.set(base, function(data){
259  chrome.tabs.query({ active: true,  currentWindow: true }, function(tabs) {
260  chrome.tabs.executeScript(tabs[0].id, { code: 'document.location.reload() ;' });
261  });
262  });
```

La query deve essere per forza all'interno della callback di `chrome.local.storage.set` in modo che anche lo script esterno possa reperire i dati della memoria locale; infatti dev'essere modificato anche l'inizio dell'esecuzione dello script esterno, prima scritto così:

```
var allPatterns ;
document.addEventListener('DOMContentLoaded', function(event) {
safari.extension.dispatchMessage('newPage');
safari.self.addEventListener('message', function(event){
  if (event.message.joojanta && event.message.joojanta.active) {
    log('Now active') ;
    allPatterns = prepare(event.message.joojanta.replacements) ;
    replaceAll(allPatterns) ;
  } else {
    log('Not active. Activate it by clicking on the purple sunglasses on your
    browser bar.') ;
  }
});
}) ;
```

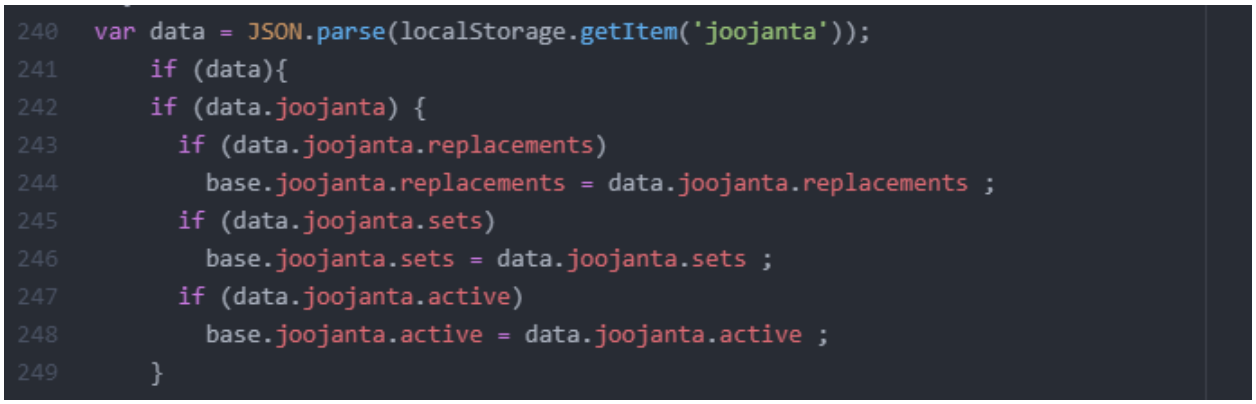
Per poi diventare così grazie a Polymorph:

```
var allPatterns ;
chrome.storage.local.get( 'joojanta', function(data) {


  if (data.joojanta && data.joojanta.active) {
    log('Now active') ;
    allPatterns = prepare(data.joojanta.replacements) ;
    replaceAll(allPatterns) ;
  } else {
    log('Not active. Activate it by clicking on the purple sunglasses on your
    browser bar.') ;
  }

}) ;
```

Stando sempre attento di chiudere i blocchi di programmazione e di modificare gli argomenti coinvolti, che passano da "`event.message`" a "`data`" in modo da evitare bug relativi a oggetti sconosciuti. Tuttavia la conversione di Polymorph non è perfetta poiché alcune istruzioni scritte in modo legittimo, in particolare le condizioni di esistenza dei dati, possono dare risultati differenti a seconda dell'interprete Javascript del browser. Ad esempio il seguente frammento di Joo Janta per Safari, convertendolo per Chrome, rimane invariato:



```
var data = JSON.parse(localStorage.getItem('joojanta'));
    if (data){
    if (data.joojanta) {
      if (data.joojanta.replacements)
        base.joojanta.replacements = data.joojanta.replacements ;
      if (data.joojanta.sets)
        base.joojanta.sets = data.joojanta.sets ;
      if (data.joojanta.active)
        base.joojanta.active = data.joojanta.active ;
    }
```

Questa è legittima, ma se si esegue l'estensione per la prima volta la condizione di esistenza su "data.joojanta" da bug che interrompono bruscamente l'interpretazione dei script Javascript dell'estensione perchè "data.joojanta" è un dato totalmente sconosciuto essendo la memoria locale in quel momento totalmente vuota, infatti l'interprete conosce "data" per la sua dichiarazione di variabile, ma non conosce, nel caso la memoria locale sia vuota, che quel dato possa contenere dei campi. Il progettista, per risolvere il problema deve aggiungere un'ulteriore condizione su data e la sintassi sarà la seguente:

```
var data = JSON.parse(localStorage.getItem('joojanta'));
if (data){
 if (data.joojanta) {
  if (data.joojanta.replacements)
   base.joojanta.replacements = data.joojanta.replacements ;
  if (data.joojanta.sets)
   base.joojanta.sets = data.joojanta.sets ;
  if (data.joojanta.active)
   base.joojanta.active = data.joojanta.active ;
 }
}
```

In questo modo anche la prima esecuzione dell'estensione avverrà senza problemi.

Capitolo 7
Conclusioni e sviluppi futuri

Polymorph, come si deduce dalla tesi, è uno strumento molto utile per il programmatore per vari motivi:

- Aiuta nella sostituzione delle istruzioni native, adattandole al modello su cui si basa il browser per il quale si desidera pubblicare un estensione.
- Offre soluzioni utili ad alcuni problemi implementativi, cercando di ridurre al minimo indispensabile il meccanismo dello scambio dei messaggi in modo da aumentare le performance dell'estensione da progettare.
- Nel caso si utilizzi l'applicazione su Joo Janta 200 seguendo le istruzioni, il progettista può visualizzare, sia con la conversione per adattarla a Chrome che con quella per adattarla a Safari, un'estensione che sfrutti le principali funzionalità dei due modelli fungendo da punto di riferimento per la progettazione ex novo di nuove estensioni grazie alla consultazione del codice sorgente prodotto.
- In particolare per le Safari App Extension, le istruzioni contengono i passaggi più importanti per effettuare una compilazione corretta dell'applicazione, evitando al programmatore lunghe ricerche riguardo i

possibili problemi riscontrati, spesso inerenti alla mancanza di integrazione delle librerie.

Tuttavia, seppur Polymorph sia molto potente, ha al tempo stesso dei limiti tali da non automatizzare totalmente il processo di conversione, in parte a causa della sintassi delle istruzioni complessa di Javascript tali da concedere molta libertà al programmatore includendo anche gli annidamenti di istruzioni, sia, nel caso delle Safari App Extension, alle procedure effettuabili solo tramite XCode per permettere una convalidazione dei file rendendo quindi possibile la pubblicazione dell'estensione sul Mac App Store.

In particolare, gli elementi da migliorare del programma sono:

- Passando da una Safari App Extension a una estensione di Chrome, l'applicazione sfrutta sia le istruzioni di memoria locale di Web Storage che quelle di Chrome, queste ultime implementate per un migliore scambio dei dati. Le versioni successive dell'applicazione, con tale conversione, implementeranno solo ed esclusivamente la memoria locale di Chrome, grazie anche a migliorie riguardo il riconoscimento dei blocchi di programmazione.
- Al momento, le istruzioni native di Chrome convertite in modo da essere eseguite sulle Safari App Extension sono poche, dato che il programma è stato pensato per convertire correttamente Joo Janta 200. Tramite test su altre estensioni, sarà possibile estendere l'insieme delle istruzioni native convertibili, spesso riguardanti una traduzione del loro codice in linguaggio Swift.
- Polymorph al momento non considera il modello delle WebExtension, standard del W3C supportato da Firefox e, in tempi recenti, Apple tramite l'ultima versione di Safari, Safari 14, disponibile per MacOS Mojave e successivi.

Polymorph osserva con interesse il futuro della storia dei browser che negli ultimi anni continua a evolversi molto velocemente. In particolare sono due gli eventi che interessano Polymorph. Il primo è il supporto delle WebExtension di Firefox e Safari 14, standard ispirato, a livello progettuale, alle estensioni di Chrome.

Tuttavia esse, oltre a non essere disponibili per i sistemi Mac con sistema operativo precedente a Mojave (rendendo necessario il mantenimento della conversione verso le Safari App Extension), necessitano comunque la creazione di un apposito progetto tramite XCode, quindi rendere un estensione Firefox disponibile per Safari rimarrà una procedura piuttosto manuale: seppur non sia necessario l'utilizzo di codice Swift, può essere necessario un riepilogo sui passaggi principali da effettuare per creare il progetto, compilarlo e distribuirlo sul Mac App Store.

L'altro evento importante riguarda Google, precisamente l'aggiornamento delle estensioni alla terza versione del file manifest.json, standard che causò anche molte critiche sia da parte dei progettisti di estensioni che da alcune aziende specializzate nello sviluppo dei browser.

La terza versione di manifest.json, detta anche Manifest V3, permette alle estensioni di aderire al nuovo standard delle estensioni di Chrome, pensate per rispettare la privacy e la sicurezza degli utenti che ne usufruiscono. Oltre a ciò, questo modello applica anche delle modifiche strutturali:

- La versione precedente delle estensioni si basa sul background page, un'interfaccia HTML con la quale l'utente può visualizzare le principali funzionalità dell'estensione. Nonostante Google consigliasse di usare gli event pages in modo che le estensioni vengano eseguite solo nel momento del bisogno, buona di parte di esse fa utilizzo di background page persistenti, tali da essere eseguite durante tutta la sessione di navigazione o fino alla disinstallazione dell'estensione, inficiando sia a livello di performance che nei possibili rischi di violazione della privacy. Manifest V3 introduce i service workers, processi che seguono la filosofia degli event page (quindi, nel caso non vengano utilizzati, non occupano inutilmente risorse) adattandola maggiormente a funzionalità che non richiedono la navigazione, di conseguenza i service workers, ad esempio, non possono eseguire operazioni riguardanti il DOM dei siti visitati.
- Le richieste di rete sono state modificate, aggiungendo un nuovo oggetto, `chrome.declarativeNetRequest`, tali anche da modificare la loro gestione, dove Chrome fa da importante intermediario per la loro

elaborazione basandosi su un insieme regole che l'estensione offre al browser.

- Viene totalmente abbandonato l'utilizzo di script remoti che riguardano, ad esempio, lo scaricamento e esecuzione di uno script Javascript proveniente da un server. D'ora in poi tutta la logica d'esecuzione dell'estensione deve essere incluso in un unico pacchetto quando viene distribuita sul Chrome Web Store. Tale scelta è mirata a migliorare la sicurezza e la loro trasparenza.
- Tra le varie modifiche figura anche la maggiore propensione al meccanismo delle promesse al posto delle callback.

Queste modifiche, alle volte sostanziali, alla struttura delle estensioni rendono l'aggiornamento delle estensioni preesistenti al nuovo modello non immediato, in particolar modo le estensioni fortemente mirate alla modifica dell'esperienza di navigazione, come le applicazioni che bloccano le pubblicità, i debugger CSS e lo stesso Joo Janta 200 usata come estensione di riferimento per la programmazione del software Polymorph.

L'introduzione di questo modello causò polemiche a causa della sua rigidità tale da limitare molte delle funzionalità precedentemente possibili, al punto che alcuni browser basati su Chromium dovettero implementare all'interno del browser un content blocker delle pubblicità. Polymoprh si interesserà a aggiungere conversioni per questi nuovi standard ormai in diffusione, offrendo in futuro la conversione per adattare l'estensione ai seguenti modelli:

- Chromium, Manifest V2, modello compatibile coi browser basati su Chromium. Questo modello è tuttora molto apprezzato per la sua semplicità combinata alle vaste possibilità implementative offerte.
- Chromium, Manifest V3, modello compatibile con Chrome e Edge. Chrome e Edge sono i browser principali tali da promuovere il nuovo modello delle estensioni di Chrome. Essendo un modello estremamente differente, pensato in particolare per le funzionalità offline, convertire un estensione a tale modello non è banale.

- Safari App Extension, per versioni precedenti di Safari 14. Questo modello verrà tenuto per questioni di compatibilità.
- WebExtension, per Mozilla Firefox e le versioni più recenti di Safari. Questo modello, solo apparentemente uguale alle estensioni basate su Manifest V2, è lo standard proposto dal W3C pensato per superare le incompatibilità coi browser. Polymorph offrirà una guida per la compilazione di questo tipo di estensioni nel caso il progettista voglia distribuirle per Safari a causa della necessità di utilizzo di XCode.

Le istruzioni di Polymorph avviseranno sempre della possibilità di problemi di esecuzione a livello, in particolare per le istruzioni condizionali, a causa dei interpreti Javascript estremamente differenti tra loro.

Bibliografia

[1] CERN, *The browser - WorldWideWeb NeXT Application*, CERN, Ultima visita: 29 Aprile 2021, https://worldwideweb.cern.ch/worldwideweb/

[2] CERN, *Line Mode Browser 2013*, CERN, Ultima visita: 29 Aprile 2021, https://line-mode.cern.ch

[3] W. Hoogland, H. Weber, *Statement Concerning CERN W3 software release into public domain*, CERN, 30 Aprile 1993, http://cds.cern.ch/record/1164399

[4] William Stewart, *Mosaic -- The first global browser*, LivingInternet.com, Ultima visita: 29 Aprile 2021, https://web.archive.org/web/20070702183017/http://www.livinginternet.com/w/wi_mosaic.htm

[5] Thom Holwerda, *The world's first graphical browser: Erwise*, OSnews, 3 Marzo 2009, https://www.osnews.com/story/21076/the-worlds-first-graphical-browser-erwise/

[6] Georgia Institute of Technology, *Graphics, visualization, and usability center - General survey results graphs*, GVU WWW User Survey, Ultima visita: 4 Maggio 2021, https://news.netcraft.com/archives/category/web-server-survey/

[7] Dr. Axel Rauschmayer, *Speaking Javascript: An in-depth guide for programmers*, Seconda edizione, O'Reilly Media, 2015, http://speakingjs.com/es5/ch04.html

[8] Tim Jackson, *This bug in your PC is a smart cookie*, Financial Times, 12 Febbraio 1996, https://archive.org/details/FinancialTimes1996UKEnglish/Feb 12 1996, Financial Times, #12, UK (en)/page/n29/mode/2up

[9] Peter Elstrom, *Microsoft's $8 million goodbye to Spyglass*, Bloomberg Businessweek, 22 Gennaio 1997, https://www.landley.net/history/mirror/ms/new0122d.htm

[10] Hakon Wium Lie, Bert Bos, *Cascading Style Sheets: Designing for the web*, terza edizione, Addison-Wesley Professional, 2005, https://www.w3.org/Style/LieBos2e/history/

[11] Microsoft Corporation, *Microsoft announces ActiveX Technologies to power the next generation of content for the Internet and PC*, Microsoft PressPass, 12 Marzo 1996, https://web.archive.org/web/20000621184123/http://www.microsoft.com/PressPass/press/1996/mar96/activxpr.asp

[12] Ed Kubaitis, *Browser statistics for June 1996*, EWS University of Illinois, Ultima visita: 29 aprile 2021, https://web.archive.org/web/20010507151448/http://www.ews.uiuc.edu/bstats/months/9606-month.html

[13] Cesare Lamanna, *CSS Crossbrowser*, HTML.it, 14 Marzo 2002, https://www.html.it/articoli/css-crossbrowser/

[14] Chuck Musciano, *The dynamic, powerful abilities of Javascript Style Sheets*, SunWorld, Aprile 1997, http://sunsite.uakom.sk/sunworldonline/swol-04-1997/swol-04-webmaster.html

[15] Forrest Stroud, *Internet Explorer 4*, WinPlanet, Ultima visita: 29 Aprile 2021, https://web.archive.org/web/20041210081033/http://cws.internet.com/article/2240-3103.htm

[16] Mary Jo Foley, *Netscape 6.0: Best… Or buggiest?*, ZDNet, 7 Novembre 2000, https://www.zdnet.com/article/netscape-6-0-best-or-buggiest/

[17] Mozilla Corporation, *Storia del progetto Mozilla*, Mozilla Corporation, Ultima visita: 29 Aprile 2021, https://www.mozilla.org/it/about/history/

[18] Alex Hopmann, *The story of XMLHTTP*, Alex Hopmann's Web Site, Ultima visita: 29 Aprile 2021, https://web.archive.org/web/20070623125327/http://www.alexhopmann.com/xmlhttp.htm

[19] Peter Bright, *Wisdom and folly: IE8's super standards mode cuts both ways*, Ars Technica, 25 Gennaio 2008, https://arstechnica.com/information-technology/2008/01/ie8-super-standards-mode/

[20] Andrew Beattie, *Why did Microsoft face antitrust charges in 1998?*, Investopedia, 30 Marzo 2020, https://www.investopedia.com/ask/answers/08/microsoft-antitrust.asp

[21] Evan Hansen, *Microsoft behind $12 million payment to Opera*, CNET, 24 Maggio 2004, https://www.cnet.com/news/microsoft-behind-12-million-payment-to-opera/

[22] British Broadcasting Corporation, *Google phases out support for IE6*, BBC News, 30 Gennaio 2010, http://news.bbc.co.uk/2/hi/technology/8488751.stm

[23] Arik Hesseldahl, *Better browser now the best*, Forbes, 29 Settembre 2004, https://www.forbes.com/2004/09/29/cx_ah_0929tentech.html?sh=7978c2d2876c

[24] Punto Informatico, *Safari 3 per Windows accolto tra i fischi*, De Andreis Editore, 14 Giugno 2007, https://www.punto-informatico.it/safari-3-per-windows-accolto-tra-i-fischi/

[25] StatCounter, *Microsoft's Internet Explorer browser falls below 50% of worldwide market for first time*, StatCounter, 5 Ottobre 2010, https://gs.statcounter.com/press/microsoft-internet-explorer-browser-falls-below-50-perc-of-worldwide-market-for-first-time

[26] W3Counter, *Global Web Stats: March 2020*, W3Counter, 31 Marzo 2020, https://www.w3counter.com/globalstats.php?year=2020&month=3

[27] Peter Bright, *Hey Presto, Opera switches to WebKit*, Ars Technica, 13 Febbraio 2013, https://arstechnica.com/information-technology/2013/02/hey-presto-opera-switches-to-webkit/

[28] Joe Belfiore, *Microsoft Edge: Making the web better through more open source collaboration*, Windows Blogs, 6 Dicembre 2018, https://blogs.windows.com/windowsexperience/2018/12/06/microsoft-edge-making-the-web-better-through-more-open-source-collaboration/

[29] Gregg Keizer, *The Brave browser basics: what it does, how it differs from rivals*, Computerworld, 8 Aprile 2021, https://www.computerworld.com/article/3292619/the-brave-browser-basics-what-it-does-how-it-differs-from-rivals.html

[30] Scott Gilbertson, *Vivaldi 2.0 review: The modern web browser does not have to be so bland*, Ars Technica, 19 Ottobre 2018, https://arstechnica.com/gadgets/2018/10/vivaldi-2-0-review-meet-your-ideal-browser-if-youre-willing-to-invest-time/

[31] W3Counter, *Global Web Stats: March 2021*, W3Counter, 31 Marzo 2021, https://www.w3counter.com/globalstats.php?year=2021&month=3

[32] W3Counter, *Web browser usage trends*, W3Counter, Ultima visita: 3 Maggio 2021, https://www.w3counter.com/trends

[33] Loren Baker, *Opera goes free with help from Google*, Search Engine Journal, 20 Settembre 2005, https://www.searchenginejournal.com/opera-goes-free-with-help-from-google/2227/#close

[34] W3Counter, *Global Web Stats: August 2017*, W3Counter, 31 Agosto 2017, https://www.w3counter.com/globalstats.php?year=2017&month=8

[35] TheCounter.com, *The full-featured web counter with graphics reports and detailed information*, TheCounter.com, Ultima visita: 7 Maggio 2021, https://web.archive.org/web/20080302002201/https://www.thecounter.com/stats/2004/April/browser.php

[36] Janet Nguyen, *Why companies still rely on Internet Explorer*, Marketplace, 26 Agosto 2020, https://www.marketplace.org/2020/08/26/why-companies-still-rely-on-internet-explorer/

[37] Claudio Davide Ferrara, *Microsoft Edge: addio a EdgeHTML*, HTML.it, 10 Dicembre 2018, https://www.html.it/10/12/2018/microsoft-edge-addio-a-edgehtml/

[38] Catalin Cimpanu, *Mozilla announces Quantum, a new browser engine for Firefox*, Softpedia News Center, 28 ottobre 2016, https://news.softpedia.com/news/mozilla-announces-quantum-a-new-browser-engine-for-firefox-509767.shtml

[39] W3Counter, *Global Web Stats: July 2010*, W3Counter, 31 Luglio 2010, https://www.w3counter.com/globalstats.php?year=2010&month=7

[40] W3Counter, *Global Web Stats: January 2016*, W3Counter, 31 Gennaio 2016, https://www.w3counter.com/globalstats.php?year=2016&month=1

[41] W3Counter, *Global Web Stats: March 2019*, W3Counter, 31 Marzo 2019, https://www.w3counter.com/globalstats.php?year=2019&month=03

[42] Mozilla Corporation, *What are extension?*, MDN Web Docs, Ultima visita: 29 Aprile 2021, https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/What_are_WebExtensions

[43] Wladimir Palant, *A not so short history of Adblock*, Adblock Plus and (a little) more - Blog, Ultima visita: 7 maggio 2021, https://web.archive.org/web/20061109142754/http://adblockplus.org/en/history

[44] Wladimir Palant, *Statistics, statistics, statistics*, Adblock Plus and (a little) more - Blog, Ultima visita: 19 Febbraio 2008, https://adblockplus.org/blog/statistics-statistics-statistics

[45] Rahul Roy-Chowdhury, *The browser for a web worth protecting*, Google Blog, 13 Febbraio 2018, https://www.blog.google/products/chrome/browser-web-worth-protecting/

[46] Janalta Interactive, *Google Toolbar*, Techopedia, Ultima visita: 7 Maggio 2021, https://www.techopedia.com/definition/27506/google-toolbar

[47] Anthony Laforge, *Saying goodbye to Flash in Chrome*, Google Blog, 25 Luglio 2017, https://www.blog.google/products/chrome/saying-goodbye-flash-chrome/

[48] Alex Perekalin, *How safe are browser extension? Things you need to know*, Kapersky Daily, 30 Gennaio 2018, https://www.kaspersky.com/blog/browser-extensions-security/20886/

[49] Lucian Constantin, *Researcher to demonstrate feature-rich malware that works as a browser extension*, Computerworld, 24 Ottobre 2012, https://www.computerworld.com/article/2492866/researcher-to-demonstrate-feature-rich-malware-that-works-as-a-browser-extension.html

[50] Microsoft Corporation, *About browser extension (Internet Explorer)*, Microsoft Docs, 15 Agosto 2017, https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa753620(v=vs.85)

[51] Erica Jostedt, *Firefox Add-ons cross more than 3 billion downloads!*, The Mozilla Blog, 26 Luglio 2012, https://blog.mozilla.org/blog/2012/07/26/firefox-add-ons-cross-more-than-3-billion-downloads/

[52] W3C Community Group, *Browser Extension Community Group Charter*, W3C Community Development Lead, Ultima visita: 29 Aprile 2021, https://browserext.github.io/charter/

[53] Erik Kay, Aaron Boodman, *Extension beta launched, with over 300 extension*, Chromium Blog, 8 Dicembre 2009, https://blog.chromium.org/2009/12/extensions-beta-launched-with-over-300.html

[54] Giuseppe Migliorino, *Estensioni per Safari 5: dove si trovano, come installarle, quali sono le migliori*, iPhoneItalia, 4 Gennaio 2011, https://mac.iphoneitalia.com/510/estensioni-per-safari-5-dove-si-trovano-come-installarle-quali-sono-le-migliori

[55] Justin Pot, *macOS Mojave will break a bunch of Safari Extensions*, How-To Geek, 11 Giugno 2018, https://www.howtogeek.com/fyi/macos-mojave-will-break-a-bunch-of-safari-extensions/

[56] Filipe Espósito, *Apple adds WebP, HDR support, and more to Safari with iOS 14 and macOS Big Sur*, 9to5Mac, 24 Giugno 2020, https://9to5mac.com/2020/06/24/apple-adds-webp-hdr-support-and-more-to-safari-with-ios-14-and-macos-big-sur/

[57] Wikimedia Foundation Inc., *Safari version history*, Wikipedia, Ultima visita: 5 Maggio 2021, https://en.wikipedia.org/wiki/Safari_version_history#Mac

[58] Apple Inc., *Licensing WebKit*, WebKit, Ultima visita: 30 Aprile 2021, https://webkit.org/licensing-webkit/

[59] Google LLC, *Chrome browser release channels*, Google Chrome Enterprise Help, Ultima visita: 30 Aprile 2021, https://support.google.com/chrome/a/answer/9027636?hl=en&ref_topic=9023448

[60] Google LLC, *Copyright 2015 The Chromium Authors. All rights reserved*, Git repositories on Chromium, Ultima visita: 30 Aprile 2021, https://www.chromium.org/Home

[61] Harshit Paul, *Chromium vs. Chrome - What's the difference*, LambdaTest, 3 Dicembre 2020, https://www.lambdatest.com/blog/difference-between-chrome-and-chromium/

[62] Jay Hoofman, *The browser engine that could*, The History of the Web, 7 Luglio 2020, https://thehistoryoftheweb.com/how-a-browser-engine-dominates-the-market/

[63] Dirk Mueller, *Greetings from the Safari team at Apple Computers*, Mailing list ARChives, 7 Gennaio 2003, https://marc.info/?m=104197092318639

[64] Clint Ecker, *Safari is first browser to pass the Acid2 test*, Ars Technica, 28 Aprile 2005, https://arstechnica.com/gadgets/2005/04/127/

[65] Maciej Stachowiak, *WebKit achieves Acid3 100/100 in public build*, Surfin' Safari, 26 Marzo 2008, https://web.archive.org/web/20080328232249/http://webkit.org/blog/173/webkit-achieves-acid3-100100-in-public-build/

[66] Philipp Lenssen, *Google Chrome, Google's browser project*, Google Blogoscoped, 1 Settembre 2008, http://blogoscoped.com/archive/2008-09-01-n47.html

[67] Apple Inc., *WebKit2*, WebKit Wiki, Ultima visita: 3 Maggio 2021, https://trac.webkit.org/wiki/WebKit2

[68] Dave Hyatt, *WebCoreRendering*, WebKit Wiki, Ultima visita: 30 Aprile 2021, https://trac.webkit.org/wiki/WebCoreRendering

[69] Peter Bright, *Google going its own way, forking WebKit rendering engine*, Ars Technica, 3 Aprile 2013, https://arstechnica.com/information-technology/2013/04/google-going-its-own-way-forking-webkit-rendering-engine/

[70] Kentaro Hara, *How Blink Works*, Google Docs, Ultima visita: 30 Aprile 2021, https://docs.google.com/document/d/1aitSOucL0VHZa9Z2vbRJSyAIsAz24kX8LFByQ5xQnUg/edit

[71] Free Code Camp Inc., *Interpreted vs compiled programming languages: What's the difference?*, FreeCodeCamp.org News, 10 Gennaio 2020, https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/

[72] OpenJS Foundation, *The V8 Javascript Engine*, Node.js, Ultima visita: 30 Aprile 2021, https://nodejs.dev/learn/the-v8-javascript-engine

[73] Andreas Link, Mario Riemann, *Big browser comparison test: Internet Explorer vs. Firefox, Opera, Safari and Chrome*, PC Games Hardware, 3 Luglio 2009, https://www.pcgameshardware.de/Tools-Software-156186/Tests/Big-browser-comparison-test-Internet-Explorer-vs-Firefox-Opera-Safari-and-Chrome-Update-Firefox-35-Final-687738/

[74] Ilya Lyamkin, *How Javascript works: Under the hood of the V8 engine*, FreeCodeCamp.org News, 26 Agosto 2020, https://www.freecodecamp.org/news/javascript-under-the-hood-v8/

[75] Apple Inc., *JavaScriptCore*, WebKit Wiki, Ultima visita: 30 Aprile 2021, https://trac.webkit.org/wiki/JavaScriptCore

[76] Aaron Boodman, *Extension Status: on the runway, getting ready for take-off*, Chromium Blog, 9 Settembre 2009, https://blog.chromium.org/2009/09/extensions-status-on-runway-getting.html

[77] Vikas SN, *The Lowdown: Google I/O 2012 day 2 - 310M Chrome users, 425M Gmail & more*, MediaNama, 29 Giugno 2012, https://web.archive.org/web/20120802073320/https://www.medianama.com/2012/06/223-the-lowdown-google-io-2012-day-2-310m-chrome-users-425m-gmail-more/

[78] Jason Kersey, *Stable channel update*, Chrome Releases, 8 Febbraio 2012, https://chromereleases.googleblog.com/2012/02/stable-channel-update.html

[79] Google LLC, *Manifest version*, Google Developer, Ultima visita: 7 Maggio 2021, https://developer.chrome.com/docs/extensions/mv2/manifestVersion/#manifest-v1-changes

[80] Amir Siddiqui, *Manifest V3 changes for browser extensions will go live in Google Chrome 88*, XDA-Developers, 9 Dicembre 2020, https://www.xda-developers.com/manifest-v3-changes-browser-extensions-go-live-google-chrome-88/

[81] Brian Weinstein, Damian Kaleta, *Extending your App with Safari App Extensions*, Apple WorldWide Developers Conference, San Francisco, 13 Giugno 2016, Apple Inc., 2016, https://developer.apple.com/videos/play/wwdc2016/214/

[82] Google LLC, *Getting started*, Chrome Developers, 28 Febbraio 2014, https://developer.chrome.com/docs/extensions/mv2/getstarted/

[83] Jake Koovor, *3 ways to get source code of any Chrome Extensions*, Saint, 25 Ottobre 2019, https://www.saintlad.com/get-source-code-of-chrome-extension/

[84] Google LLC, *Developer program policies*, Chrome Developers, 28 Febbraio 2014, https://developer.chrome.com/docs/webstore/program_policies/?csw=1#extensions

[85] Google LLC, *Extensions quality guidelines FAQ*, Chrome Developers, 9 Maggio2014, https://developer.chrome.com/docs/extensions/mv2/single_purpose/

[86] Apple Inc., *Building a Safari App Extension*, Apple Developer, Ultima visita: 30 Aprile 2021, https://developer.apple.com/documentation/safariservices/safari_app_extensions/building_a_safari_app_extension

[87] Apple Inc., *Converting a Legacy Safari Extension to a Safari App Extension*, Apple Developer, Ultima visita: 30 Aprile 2021, https://developer.apple.com/documentation/safariservices/safari_app_extensions/converting_a_legacy_safari_extension_to_a_safari_app_extension

[88] Brian Donohue, *Converting Legacy Safari Extensions*, Brian Donohue - Makin' that Instapaper, 12 Dicembre 2019, https://bthdonohue.com/2019/12/12/converting-legacy-safari-extensions.html

[89] Giuseppe Sapienza, *Le novità di XCode8 e Swift 3. Semplicità, la nuova parola d'ordine*, xCoding.it, 13 Settembre 2016, https://www.xcoding.it/le-novita-di-xcode-8-swift-3/

[90] Apple Inc., *App Extensions programming guide: Understand how an App Extension works*, Apple Developer, Ultima visita: 30 Aprile 2021, https://developer.apple.com/library/archive/documentation/General/Conceptual/ExtensibilityPG/ExtensionOverview.html

[91] Apple Inc., *Passing Messages between Safari App Extensions and injected scripts*, Apple Developer, Ultima visita: 30 Aprile 2021, https://developer.apple.com/documentation/safariservices/safari_app_extensions/passing_messages_between_safari_app_extensions_and_injected_scripts

[92] Bart Jacobs, *iOS da zero con Swift: Swift in breve*, Envato Tus+, 9 Dicembre 2015, https://code.tutsplus.com/it/tutorials/ios-from-scratch-with-swift-swift-in-a-nutshell--cms-25168

[93] Abhilash KM, *Type casting in Swift: difference between is, as, as?, as!*, Medium, 14 Luglio 2017, https://abhimuralidharan.medium.com/typecastinginswift-1bafacd39c99