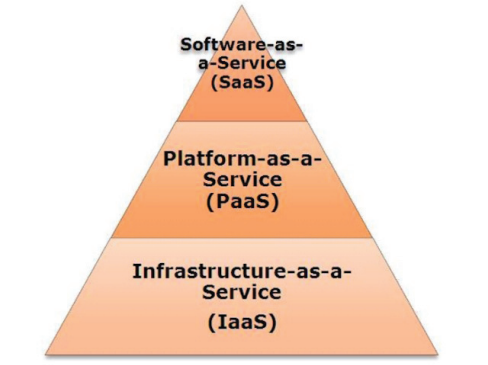
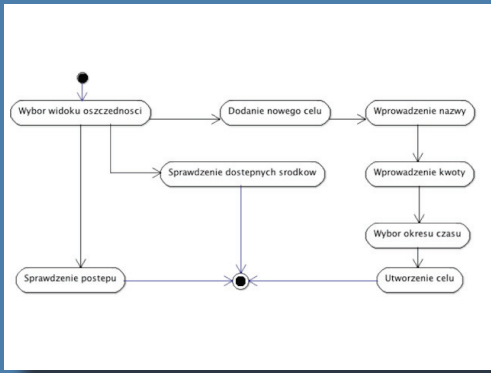
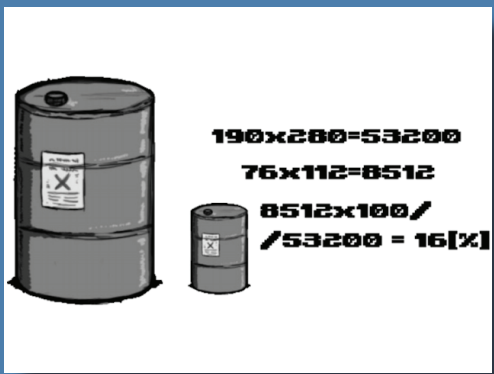
Problemy Współczesnej Inżynierii
Technologie programistyczne





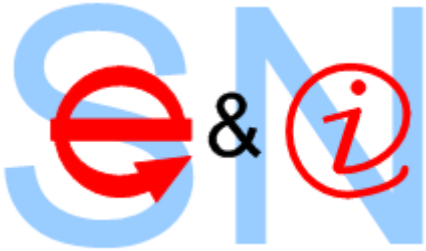


redakcja
Maciej Laskowski
Paweł A. Mazurek
Tomasz N. Kołtunowicz
Piotr Z. Filipek

Politechnika Lubelska
Lublin 2014

Problemy Współczesnej Inżynierii

Technologie programistyczne



Politechnika Lubelska
Wydział Elektrotechniki i Informatyki
ul. Nadbystrzycka 38A
20-618 Lublin

Problemy Współczesnej Inżynierii

Technologie programistyczne

redakcja:
Maciej Laskowski
Paweł A. Mazurek
Tomasz N. Kołtunowicz
Piotr Z. Filipek

Politechnika Lubelska
Lublin 2014

Recenzenci:
członkowie Komitetu Naukowego IV Sympozjum Elektryków i Informatyków

Skład: Maciej Laskowski

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

ISBN: 978-83-7947-068-6

Wydawca: Politechnika Lubelska
ul. Nadbystrzycka 38D, 20-618 Lublin
Realizacja: Biblioteka Politechniki Lubelskiej
Ośrodek ds. Wydawnictw i Biblioteki Cyfrowej
ul. Nadbystrzycka 36A, 20-618 Lublin
tel. (81) 538-46-59, email: wydawca@pollub.pl
www.biblioteka.pollub.pl
Druk: TOP Agencja Reklamowa Agnieszka Łuczak
www.agencjatop.pl

Elektroniczna wersja książki dostępna w Bibliotece Cyfrowej PL www.bc.pollub.pl
Nakład: 100 egz.

SPIS TREŚCI

PRZEDMOWA

Szanowni Uczestnicy i Sympatycy Sympozjum, Czytelnicy

Oddana w Państwa ręce monografia jest opracowaniem naukowym zawierającym wybrane zrecenzowane referaty wygłoszone na IV Sympozjum Naukowym Elektryków i Informatyków SNEiI2014, które kolejny raz odbyło się z inicjatywy studentów, członków i opiekunów kół naukowych zrzeszonych na Politechnice Lubelskiej oraz Samorządu Studenckiego w dniach 6–7 marca 2014 na Wydziale Elektrotechniki i Informatyki Politechniki Lubelskiej.

Celem tego naukowego wydarzenia, była i jest, wymiana informacji i doświadczeń we wskazanych obszarach wśród młodej społeczności akademickiej oraz przedstawicieli władz miasta i przemysłu w regionie. Nowością było zorganizowanie cyklu warsztatów dla studentów – członków kół naukowych z zakresu informatyki oraz programowania sterowników.

Zakładany przez Komitet Organizacyjny Sympozjum zakres rozważań był i jest szeroki, bowiem pokrywa problematykę teorii i zastosowań elektrotechniki, mechatroniki oraz informatyki w badaniach naukowych, edukacji i przemyśle. Znaczna liczba artykułów pokonferencyjnych i ich wysoka jakość merytoryczna ponownie zmobilizowała nas do opublikowania monografii w wersji dwutomowej, rozdzielonej tematycznie tym razem na Technologie Programistyczne, Technologie Informacyjne oraz Elektrotechnologie.

Jako redaktorzy monografii pokonferencyjnych i współorganizatorzy Sympozjum cieszymy się, że niniejsza inicjatywa ciągle wykazuje bardzo pozytywny odbiór zarówno środowiska akademickiego, jak i lubelskiego przemysłu. Tak jak w poprzednich wydarzeniach aktywnie uczestniczyli w naszym sympozjum przedstawiciele regionalnych i lubelskich uczelni – Uniwersytetu Medycznego, Uniwersytetu Przyrodniczego, Katolickiego Uniwersytetu Lubelskiego, Wyższej Szkoły Ekonomii i Innowacji oraz Państwowej Wyższej Szkoły Zawodowej w Chełmie.

Patronat honorowy nad naszym Sympozjum objął ponownie Prezydent Miasta Lublin – dr Krzysztof Żuk, pani Prezes Urzędu Komunikacji Elektronicznej – Magdalena Gaj oraz Lubelski Oddział Stowarzyszenia Elektryków Polskich.

W gronie patronujących instytucji znalazło się także Polskie Towarzystwo Informatyczne – Koło w Lublinie oraz Lubelski Oddział Polskiego Towarzystwa Zarządzania Produkcją. Sympozjum oraz publikacje pokonferencyjne mogły zaistnieć dzięki wsparciu Jego Magnificencji Rektora Politechniki Lubelskiej prof. dr hab. inż. Piotra Kacejki oraz Pani Dziekan Wydziału Elektrotechniki i Informatyki prof. dr hab. inż. Henryki D. Stryczewskiej. Serdeczne podziękowania kierujemy także do firm, które sponsoringiem wsparły materialnie i rzeczowo nasze Sympozjum, a firmie WAGO szczególnie dziękujemy za przeprowadzenie dwudniowego cyklu szkoleń z programowania sterowników PLC.

Szczególne podziękowania składamy prelegentom – Sławomirowi Cybulskiemu z ABB sp. z o.o., Mateuszowi Żarkowskiemu z Politechniki Wrocławskiej oraz Pawłowi Prokopowi z Grupy Projektowej KONCEPT, którzy w odpowiedzi na zaproszenie Lubelskiego Oddziału SEP oraz przedstawicieli Wydziału, przedstawili referaty w sesji otwartej. Podziękowania składamy również panu Michałowi Furmankowi z Departamentu Rozwoju i Obsługi Inwestorów Urzędu Miasta Lublin za pomoc oraz prezentację multimedialną prezentującą wsparcie inicjatyw akademickich przez władze miasta.

W trakcie Sympozjum studenci i uczestnicy mogli też zapoznać się z prezentacją Ruchomej Stacji Pomiarowej wykorzystywanej przez Delegaturę Urzędu Komunikacji Elektronicznej w Lublinie, a także mieli możliwość uczestniczenia w pokazie minipaneli podzespołów pneumatyki prowadzonym przez firmę SMC.

Jako organizatorzy będziemy się starać, aby taka forma prezentacji naukowych i stanowiskowych osiągnięć studentów i kół naukowych stała się cyklicznym wydarzeniem, ważnym i ciekawym w procesie dydaktycznym i życiu akademickim.

Zachęcamy uczestników Sympozjum i Czytelników do zapoznania się ze wszystkimi tomami monografii pokonferencyjnych - technologiami programistycznymi, technologiami informacyjnymi oraz elektrotechnologiami.

Z życzeniami miłej lektury,

Paweł A. Mazurek
Maciej Laskowski
Tomasz N. Kołtunowicz
Piotr Z. Filipek

KAROL BANASZKIEWICZ[1], EDYTA ŁUKASIK[2]

PRZETWARZANIE ZDJĘĆ Z WYKORZYSTANIEM CHMURY

WSTĘP

Branża IT zmienia się z bardzo szybkim tempie. Jedną z najnowszych jej dziedzin są chmury obliczeniowe. Powszechnie wiadomo, że chmury obliczeniowe są przyszłościową technologią. W jednym z najnowszych raportów firma Gartner prezentuje ciągle poszerzające się wykorzystanie tej technologii w wielu aspektach biznesowych [5].

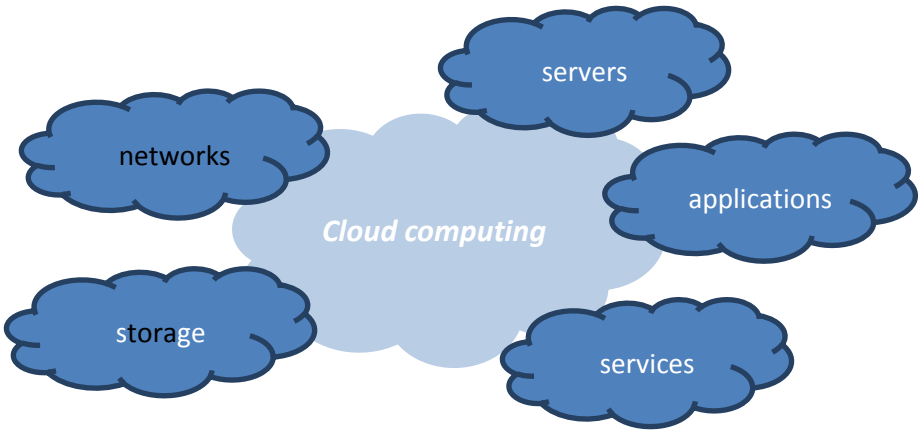
Przetwarzanie w chmurze gwarantuje wysoką wydajność działania aplikacji na niej uruchomionej. Właśnie z tego powodu tak wiele firm tworzy nowe projekty w oparciu o nią, a nawet przenosi tam istniejące wewnętrzne rozwiązania. Dostawcy chmur prześcigają się w zapienianiu bezpieczeństwa, udoskonalaniu istniejących oraz dodawaniu nowych usług. Często proponują też własne innowacyjne podejście do tematu chmur obliczeniowych, co w znacznym stopniu przyczynia się do gwałtownego rozwoju tej dziedziny informatyki.

Zastosowanie chmury obliczeniowej w połączeniu z aplikacją na urządzenia mobilne, które obecnie także zdominowały rynek, niewątpliwie jest godne uwagi. Zaprezentowana zostanie aplikacja, która zapewni wysoką wydajność pomimo małej mocy urządzenia, które będzie z niej korzystało. Połączona zostanie w ten sposób wysoka wydajność, jaką niesie ze sobą chmura obliczeniowa, z niemal nieograniczoną mobilnością telefonu komórkowego. Zaprezentowana aplikacja oparta jest o chmurę Windows Azure oraz platformę Windows Phone 8 i pozwalają ona na zdalną obróbkę zdjęć. Zostanie także zbadana wydajność przetwarzania obrazów w chmurze.

[1]Politechnika Lubelska, Wydział Elektrotechniki i Informatyki,
student kierunku Informatyka
[2] Politechnika Lubelska, Wydział Elektrotechniki i Informatyki, Instytut Informatyki

CZYM JEST CHMURA OBLICZENIOWA?

Definicji tego pojęcia jest dość dużo. Jedną z najczęściej używanych jest jednak ta podana przez NIST (National Institute of Standards and Technology): „Chmura obliczeniowa to model umożliwiający powszechny, wygodny, udzielany na żądanie dostęp za pośrednictwem sieci do wspólnej puli możliwych do konfiguracji zasobów przetwarzania (np. sieci, serwerów, zasobów przechowywania, aplikacji i usług), które można szybko dostarczyć i uwolnić przy minimalnym wysiłku zarządzania lub działania ze strony usługodawcy”[8]. Cechą charakterystyczną wynikającą z tej, jak i wielu innych powszechnie używanych definicji dla chmury, jest to, że zasoby udostępniane są w postaci usług. Nie ma tutaj znaczenia czy mowa o mocy obliczeniowej, przestrzeni dyskowej czy aplikacji. Budowa chmury obliczeniowej została zaprezentowana na Rys. 1.



Rys. 1. Chmura obliczeniowa
(źródło: opracowanie własne)

Do najczęściej wymienianych zalet chmury obliczeniowej należą [4,13]:

- redukcja kosztów – brak potrzeby zakupu sprzętu oraz zatrudnienia administracji IT;
- skalowalność – firma płaci tylko za te zasoby, z których faktycznie korzysta;
- równość małych i dużych firm – licencje na nowe oprogramowanie są znacznie tańsze w chmurach, przez co nawet małe firmy mogą pracować na najnowszych rozwiązania;

- odciążenie lokalnych serwerów – usługi wymagające dużych zasobów mogą zostać przeniesione do chmury;
- łatwy dostęp – usługa jest dostępna z każdego miejsca mającego połączenie z Internetem.

Zasada działania chmur w przypadku każdego z jej dostawców wygląda w ten sam sposób. Duże, składające się z setek serwerów, centra przetwarzania danych kontrolowane są przez systemy operacyjne zapewniające wirtualizację. Dla konsumenta oznacza to, że ich aplikację są uruchamiane wraz z innymi na tym samym serwerze, ale w żaden sposób ze sobą nie kolidują. Ważną zaletą chmur jest jej niemal nieograniczona moc i pojemność, co w połączeniu ze skalowalnością pozwala na wykupienie taki zasobów, jakich aktualnie potrzebuje klient. Jeśli aplikacja będzie wymagała większej mocy obliczeniowej, wystarczy zmienić parametry chmury i płacić o kilka złotych miesięcznie więcej, bez konieczności faktycznego dokupowania sprzętu do serwerowni[3].

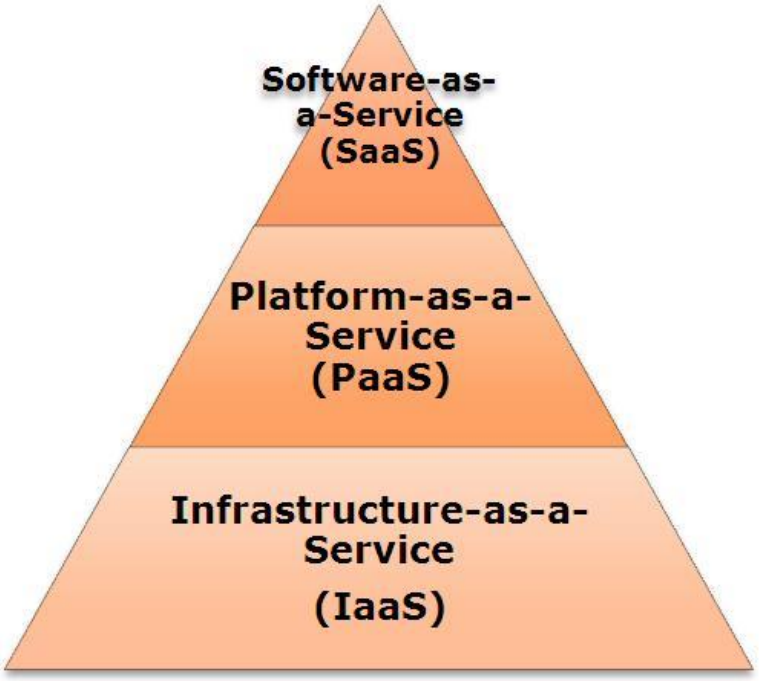
Chmury obliczeniowe można podzielić na:

- prywatne – jest to rozwiązanie przeznaczone dla jednej organizacji będące częścią jej wewnętrznej struktury;
- publiczne – ogólnie dostępna usługa w ramach globalnych zasobów;
- hybrydowa – połączenie koncepcji chmury prywatnej i chmury publicznej. Część aplikacji pracuje w chmurze prywatnej, a część w obszarze chmury publicznej[12].

Podział chmur ze względu na poziom świadczonych usług to:

- IaaS – infrastruktura jako usługa;
- PaaS – platforma jako usługa;
- SaaS – oprogramowanie jako usługa [6].

Na Rys. 2 został przedstawiony ten podział jako piramida, w której najniżej znajduje się infrastruktura jako usługa – Infrastructure as a Service (IaaS), wyżej jest platforma jako usługa – Platform as a Service(PaaS), a na samym szczycie znajduje się oprogramowanie jako usługa – Software as a Service(SaaS).



Rys. 2. Modele chmur obliczeniowych
(źródło: [10])

SaaS z kilku powodów jest podobny do aplikacji starego modelu "cienkiego klienta", gdzie klient, w tym przypadku najczęściej przeglądarka internetowa, pełni rolę punktu dostępu do oprogramowania uruchomionego na serwerze. SaaS jest najbardziej znaną dla konsumentów formą świadczenia usług z wykorzystaniem chmury obliczeniowej. Pozwala ona na przeniesienie zadań związanych z zarządzaniem oprogramowaniem i jego wdrożeniem na barki dostawcy chmury. Pośród najbardziej rozpoznawalnych aplikacji opartych o ten model znajdziemy: aplikacje do zarządzania relacjami z klientem takie jak Salesforce, pakiet biurowy np. Google Apps oraz aplikacje odpowiedzialne za przechowywanie danych: Box lub DropBox. Wykorzystanie aplikacji SaaS przyczynia się do zmniejszenia kosztów eksploatacji oprogramowania poprzez zniesienie potrzeby instalacji, zarządzania, aktualizowania aplikacji oraz zredukowania kosztów związanych z licencjonowanie oprogramowania. Aplikacje SaaS są zwykle świadczone w modelu subskrypcji.

PaaS znajduje się niżej w hierarchii niż SaaS. Zazwyczaj stanowi ona platformę, na której oprogramowanie może zostać stworzone i wdrożone. Dostawca PaaS zapewnia serwer z systemem operacyjnym i oprogramowaniem serwerowym oraz gwarantuje opiekę nad serwerem i sieciową infrastrukturą. Klient ma swobodę w decydowaniu o wyglądzie i funkcjonowaniu własnego produktu lub usługi. Tak jak ma to miejsce w przypadku większości usług w chmurze, PaaS jest zbudowany na technologii wirtualizacji. Firmy mogą zamawiać zasoby według aktualnych potrzeb, dowolnie je skalując w razie zwiększonego zapotrzebowania. Przykładami dostawców PaaS są Heroku, Google App Engine i Red Hat's OpenShift.

Podstawowym modelem chmury jest IaaS. Składa się on z wysoce zautomatyzowanych i skalowalnych zasobów obliczeniowych, uzupełnionych o możliwość składowania danych, które mogą być odmierzone i dostępne na żądanie. Dostawcy IaaS oferują serwery w chmurze i związane z nimi zasoby poprzez odpowiednie API. Klienci IaaS mają bezpośredni dostęp do swoich serwerów i pamięci masowej, tak samo, jak ma to miejsce w przypadku tradycyjnych serwerów, ale ze znacznie rozbudowaną opcją skalowalności. Użytkownicy IaaS mogą zlecić stworzenie "wirtualnego centrum danych" w chmurze i mieć dostęp do wielu z tych samych technologii i możliwości jak w przypadku tradycyjnego centrum danych, bez konieczności inwestowania w planowanie i utrzymanie fizycznych urządzeń. Jest to najbardziej elastyczny z modeli chmur obliczeniowych i umożliwia zautomatyzowane wdrażanie serwerów, dostęp do mocy obliczeniowej, przechowywania danych i sieci. Główne zastosowania IaaS obejmują rzeczywisty proces wytwarzania i wdrażanie aplikacji PaaS, SaaS oraz skalowalnych aplikacji internetowych. Istnieje wielu dostawców oferujących infrastruktura jako usługa, takich jak CloudSigma, HPCloud i SOFTLAYER [3].

DZIAŁANIE I ARCHITEKTURA CHMURY

Istotą chmury, i to każdej bez wyjątku, jest jej usługowy charakter. Istnieją dwie technologie, które odgrywają kluczową rolę w pracy z chmurami:

- wirtualizacja;
- architektura zorientowana na serwisy.

Współczesne chmury i przetwarzanie sieciowe posiadają jednak bardzo istotny wspólny element, jakim jest zdolność korzystania z zasobów rozproszonych. Należy jednak pamiętać o różnym sposobie alokowania tych zasobów i wynikającej z tego skalowalności.

Skoro cechą podstawową chmury jest jej usługowy charakter, to oferowany za jej pomocą model przetwarzania danych musi uwzględniać zróżnicowane zapotrzebowanie ze strony potencjalnych użytkowników.

Oznacza to, że firma będąca dostawcą chmury powinna nie tylko dysponować mocą obliczeniową i odpowiednią przepustowością łącza, ale również zapewniać elastyczne dostosowanie możliwości infrastruktury do potrzeb klientów.

Aby tego dokonać konieczna jest wirtualizacja posiadanych zasobów.

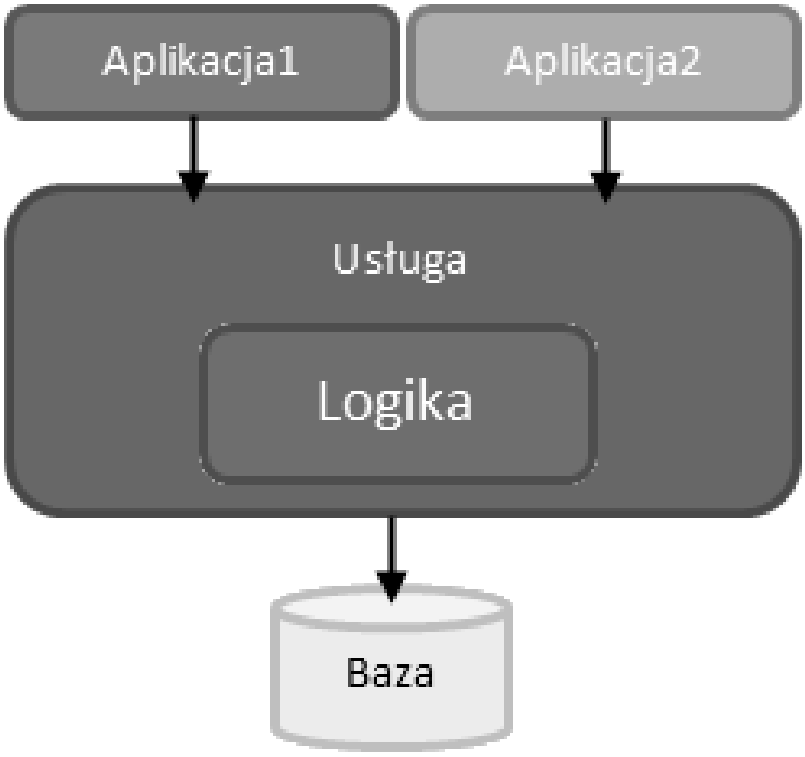
Nie należy tego mylić z pojęciem wirtualnego superkomputera czy też klastra. W teorii wirtualizacja pozwala utworzyć dowolną liczbę maszyn wirtualnych, na których mogą być uruchomione dowolne zadania i usługi. W praktyce ograniczenie stanowią dostępne zasoby, czyli pamięć operacyjna czy też moc obliczeniowa. Są to jednak jedynie ograniczenia ilościowe, a nie narzucone szczegółami konstrukcyjnymi czy infrastrukturą, bo wirtualizacja pozwala na utworzenie dowolnej, dostosowanej do wymagań klienta maszyny wirtualnej. Maksymalizując wydajność poszczególnych serwerów wirtualizacja zmniejsza koszty związane z zakupem większej liczby fizycznych maszyn[2]. Dostawca chmury potrzebuje przynajmniej dwa razy więcej miejsca, niż wynika to z faktycznego zapotrzebowania klientów. Dzieje się tak dlatego, że dyski tak samo jaki komputery i inne urządzenia potrafią ulec awarii. Dostawca musi więc stworzyć kopie danych klientów i zapisać ją w innym miejscu

Kolejną kluczową technologią w pracy z chmurami jest architektura zorientowana na serwisy (SOA). Jest to koncepcja tworzenia systemów informatycznych, w której główny nacisk stawia się na definiowanie usług, które spełnią wymagania użytkownika. Otwarta architektura przetwarzania w chmurze definiuje siedem podstawowych zasad, na jakich powinna opierać się chmura i znajdująca się w niej aplikacja[9]. Są to[1]:

- zintegrowane zarządzanie ekosystemem chmury;
- wirtualizacja – chodzi o wirtualizację sprzętową, która polega na zarządzaniu urządzeniami fizycznymi w trybie plug-and-play oraz wirtualizację oprogramowania, czyli użycie programu do zarządzania obrazami oprogramowania w celu umożliwienia dzielenia się aplikacjami;
- architektura zorientowana na usługi (SOA);

- elastyczna rezerwacja zasobów;
- usługi jako kompletne produkty;
- jednolita reprezentacja informacji;
- jakość i porządek.

SOA jest podejściem, które ma na celu odizolowanie warstwy logiki od warstwy prezentacji. Takie podejście pozwala na rozwiązanie wielu problemów w przeciwieństwie do podejścia warstwowego, w którym jeżeli dwie aplikacje używają tej samej bazy oraz posiadają pewną logikę pozwalającą na operacje na danych w niej zawartych w przypadku potrzeby zmian w logice, wymagana będzie aktualizacja obu aplikacji. Schemat architektury zorientowanej na serwisy przedstawiony został na Rys. 3. Widać tutaj wyraźne oddzielenie warstwy logiki od bazy danych.



Rys. 3. Schemat SOA
(źródło: [11])

PORÓWNANIE DOSTAWCÓW CHMUR

W Tabeli 1 podano zestawienie usług i wsparcie dla różnych technologii w każdej z analizowanychchmur. Zawarte są w niej najważniejsze z punktu widzenie dewelopera szczegóły, które zostały podzielone na cztery kategorie: wspierane frameworki i języki programowania, dostępne bazy danych, metody developmentu oraz wsparcie.

Tabela 1. Zestawienie cech dostawców chmur obliczeniowych [7]

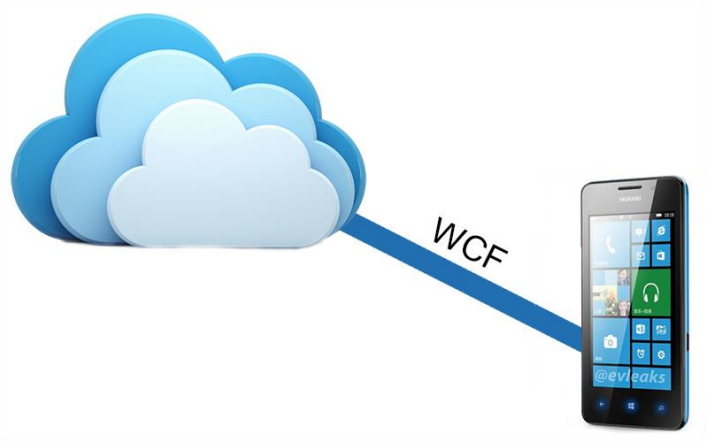
	Heroku	**Amazon**	**Windows Azure**	**AppFog**
Wspierane frameworki i języki programowania				
PHP	+	+	+	+
Python	+	+	+	+
Ruby	+	+	-	+
ASP.NET	-	+	+	-
Java	+	+	+	+
Django	+	+	+	+
Rails	+	+	-	+
Node.js	+	+	+	+
WordPress	+	+	+	+
Drupal	+	+	+	+
Joomla	+	+	+	+
Bazy Danych				
PostgreSQL	+	-	-	+
MySQL	-	+	+	+
MongoDB/NoSQL	+	+	+	+
Blobs/Binary Storage	+	-	+	-
Opcje developmentu				
Command-line Interface	+	+	+	+
Git	+	+	+	+
FTP	-	-	+	+
Wsparcie				
Dokumentacja i FAQ	+	+	+	+
Email	-	-	+	+
Phone	-	-	+	-

Pod względem wspierania frameworków i języków jest w każdej z chmur bardzo dogodna sytuacja. Trzy z porównywanych chmur obsługują trzy bazy danych, a tylko jedna dwie. Dwie z nich nie obsługują ftp i tylko jedna daje pełne wsparcie dla email i phone. Chmurą, która posiada wsparcie dla aplikacji mobilnych jest Windows Azure. Została ona wybrana do przetestowania.

APLIKACJA KORZYSTAJĄCA Z ZASOBÓW CHMURY

Stworzona została aplikacja mobilna umożliwiająca obróbkę zdjęć w chmurze za pośrednictwem telefonu. Tego typu rozwiązanie pozwala na zaawansowane i często wymagające dużej mocy obliczeniowej operacje na zdjęciach bez użycia drogich i rozbudowanych telefonów. Do takiej obróbki zdjęć wystarczy dowolny telefon z połączeniem do Internetu i systemem Windows Phone 8.

Wszystkie operacje będą wykonywane w chmurze, a ich wyniki odsyłane na urządzenie mobilne. Aplikacja składa się z dwóch modułów. Jeden moduł, którego zadaniem jest udostępnianie usług za pomocą web serwisów, znajduje się w chmurze Azure. Drugą część stanowi aplikacja na platformę Windows Phone 8, która korzysta z wystawianych usług.
Działanie aplikacji zaprezentowane jest na Rys. 4.



Rys. 4. Współdziałanie aplikacji z chmurą
(źródło: opracowanie własne)

Jedyną dostępną w tej chwili opcją jest przesłanie zdjęcia do chmury. Najpierw należy wybrać obraz, a następnie zostanie on przesłany. Do tego celu użyto klasy *PhotoChooserTask* z pakietu *Microsoft.Phone.Tasks*.

Na Listingu 1 zaprezentowana została funkcja, która odpowiada za wybór zdjęcia. Funkcja ta odpowiada za sprawdzenie, czy został wybrany jakiś obraz. Następnie z wykorzystaniem klasy *WriteableBitmap* zapisuje go ona do tablicy bajtów i w takiej formie wysyła na serwer.

Listing 1. Obsługa zdarzenia wyboru obrazu (źródło: opracowanie własne)

```
1   void photoChooserTask_Completed(object sender, PhotoRes
    ult e)
2   {
3      if (e.TaskResult == TaskResult.OK)
4      {
5          var bitmapImage = new BitmapImage();
6          bitmapImage.SetSource(e.ChosenPhoto);
7          var wBitmap = new WriteableBitmap(bitmapImage);
8          var ms = new MemoryStream();
9          wBimap.SaveJpeg(ms, wBitmap.PixelWidth,
                      wBitmap.PixelHeight, 0, 100);
10         Glob-
    al.serviceClient.uploadImageAsync(ms.ToArray());
11         Global.wait = true;
12         pleaseWaitChecker();
13     }
14  }
```

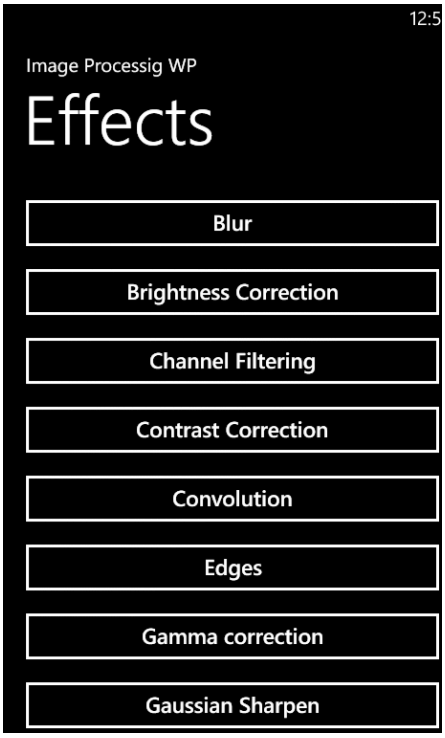
Po wgraniu obrazu na serwer opcja edycji staje się dostępna, a po kliknięciu w nią nawigacja zostaje przekazana do nowego okna z listą dostępnych efektów, którym może zostać poddany obraz. Spis dostępnych do realizacji efektów przedstawiony został na Rysunkach 5a i 5b.

Po stronie aplikacji mobilnej obsługa każdego z efektów wygląda niemal identycznie. Funkcja odpowiedzialna za realizację efektu uwypuklenia (*convolution*) została przedstawiona na Listingu 2.

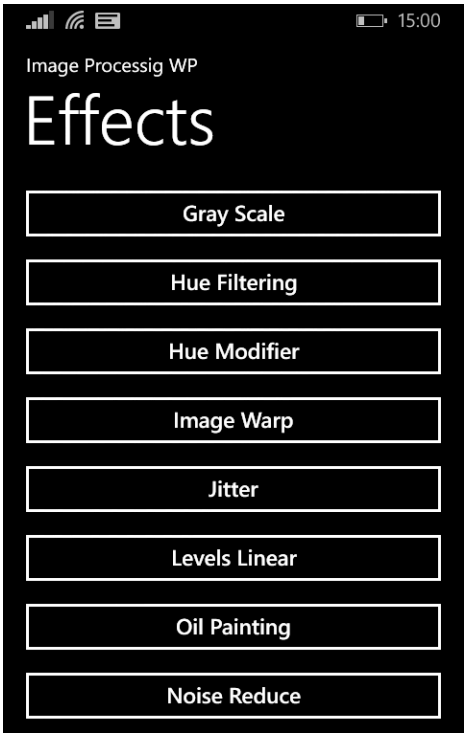
Listing 2. Obsługa efektu po stronie aplikacji mobilnej (źródło: opracowanie własne)

```
1   private void convolution_button_Click(object sender,
                RoutedEventArgs e)
2   {
3          Global.wait = true;
4          Global.serviceClient.convolutionAsync (Glob-
    al.actualImage.current);
5          NavigationService.GoBack();
6   }
```

Po wybraniu efektu jest on realizowany w chmurze i tam wywołana zostaje funkcja serwera odpowiedzialna za dany efekt przekazując id obrazka, na który ma zostać nałożony efekt i następuje w aplikacji powrót do ekranu głównego. W tym czasie w głównym oknie wyświetlona jest wiadomość o oczekiwaniu na odpowiedź serwera. Następnie na ekranie telefonu użytkownik ma wyświetlony obraz po zastosowanym efekcie.



Rys. 5a. Efekty w aplikacji
(źródło: opracowanie własne)



Rys. 5b. Efekty w aplikacji – c.d.
(źródło: opracowanie własne)

Na Listingu 3 przedstawiona została metoda, która jest wywoływana po zakończeniu pracy w chmurze. Kiedy obraz zostanie już zmieniony w zamierzony przez użytkownika sposób, istnieje możliwość zapisana go w oryginalnym rozmiarze w pamięci telefonu. Wystarczy wybrać odpowiednią ikonę w aplikacji.

Listing 3. Reakcja na zakończenie operacji na serwerze (źródło: opracowanie własne)

```
1  void serviceClient_convolutionCompleted(object sender,
       convolutionCompletedEventArgs e)
2  {
3      var r = e.Result.image;
4      Global.actualImage = e.Result;
5      set_image(r);
6  }
```

Aplikacja mobilna jest tylko klientem korzystającym z usług udostępnianych przez serwer. To właśnie w chmurze przechowywane są dane oraz wykonywane wszystkie, kosztowne obliczeniowo operacje. Dzięki wykorzystaniu Windows Azure SDK proces tworzenia aplikacji przeznaczonej dla chmur staje się o wiele łatwiejsze. Struktura aplikacji znajdującej się w chmurze jest bardzo prosta. W chmurze uruchomiona jest aplikacja WCF Service, czyli aplikacja zorientowana na udostępnianie usług. Aby uzyskać większą przejrzystość kodu, cały mechanizm, odpowiedzialny za składowanie oraz przetwarzanie obrazów, został przeniesiony do oddzielnego projektu, który jest kompilowany do postaci pliku dll i w takiej formie wykorzystany w aplikacji WCF jako biblioteka.

Nastąpiło więc oddzielenie warstwy logiki od warstwy komunikacji z użytkownikiem. Podstawową z udostępnionych przez aplikację usług jest możliwość przesłania obrazu do chmury. Operacja te wykonywana jest za pomocą funkcji *uploadImage,* której implementację przedstawiono na Listingu 4. Funkcja ta otrzymuje od klienta tablicę bajtów reprezentującą przesłany obraz i przesyła ją dalej do funkcji obiektu logiki odpowiedzialnej za zapisanie obrazu. Następnie dokonuje konwersji otrzymanego obiekt *ImageResponse* z klasy *ProcessingLogic* zawierającego kilka dodatkowych pól na ten, który powinien zostać przesłany do klienta.

Listing 4. Metoda wgrania obrazu na serwer (źródło: opracowanie własne)

```
1   ProcessingLogic.ProcessingLogic pl = new ProcessingLo-
    gic.ProcessingLogic();
2
3   public ImageResponse uploadImage(byte[] composite)
4   {
5       return toServiceRe-
6   sponse(pl.uploadImage(composite));
7   }
8
    public ImageResponse toServiceRe-
9   sponse(ProcessingLogic.ImageResponse ir)
10  {
11   ImageResponse response = new ImageResponse();
12      response.current = ir.current;
13      response.image = ir.scaledImage;
14      response.next = ir.next;
15      response.previous = ir.previous;
16      return response;
17  }
```

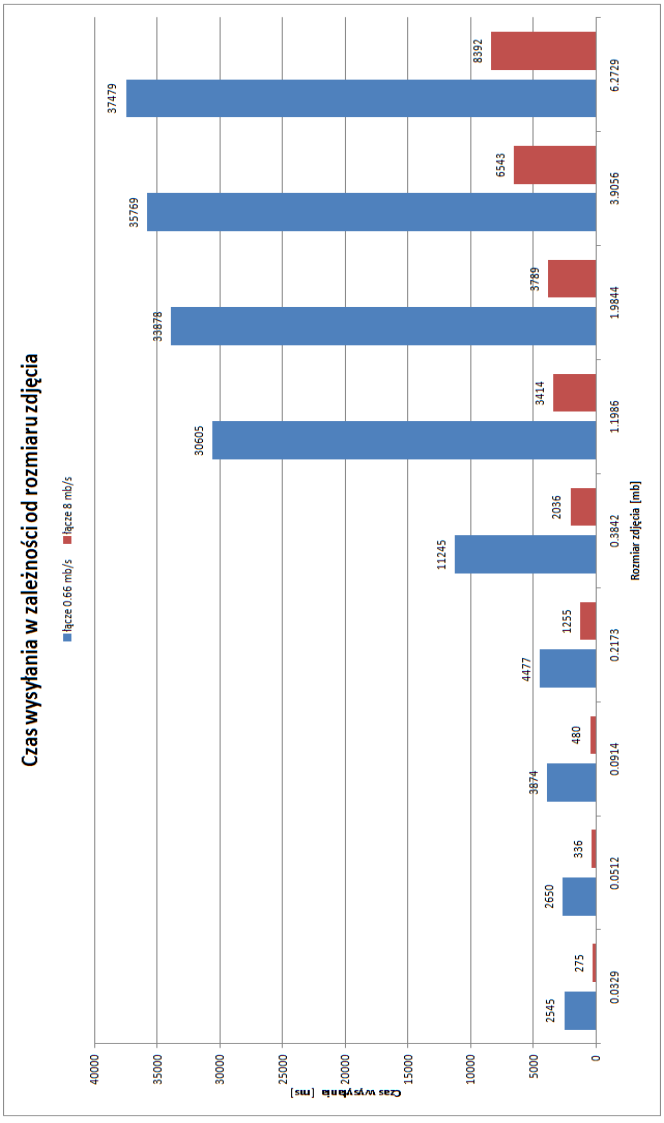
Opisany powyżej fragment kodu stanowi jedynie warstwę pośrednią pomiędzy aplikacją kliencką, a właściwą logiką serwerową. Najważniejsza jest tutaj funkcja *uploadImage* wywołana na obiekcie logiki. Otrzymana tablica bajtów jest konwertowana do obiektu Bitmapy i zapisywana w odpowiednim formacie. Tutaj jest to *Format32bppRgb*.

Kolejny krok to stworzenie obiektu, który będzie przechowywał odpowiednie dane, a następnie wywołanie funkcji w celu przeskalowania obrazu. Z wykorzystaniem filtru *ResizeBilinear* obraz skalowany jest w taki sposób, że jego szerokość wynosi 480 pikseli, przy jednoczesnym zachowaniu oryginalnych proporcji. Takie ustawienie rozmiaru obrazka pozwala znacznie zmniejszyć liczbę przesyłanych do klienta bajtów przy jednoczesnym zachowaniu ostrości obrazu. Szerokość 480 pikseli posiada większość wyświetlaczy telefonów z systemem Windows Phone 8. Przeskalowany obraz wpisany zostaje w formie tablicy bajtów. Obraz ten nie będzie poddawany już żadnej obróbce, a jedynie odsyłany w odpowiedzi na żądania aplikacji, dlatego nie jest konieczne przechowywanie go w formie Bitmapy.

Kompletny już obiekt dodany zostaje do listy obrazów zastępującej w aplikacji bazę danych. Dzięki użyciu listy przechowywanej w pamięci niwelowany zostaje problem niepotrzebnych opóźnień związanych z komunikacją z bazą danych. Użytkownik może przeglądać historię obrazów.

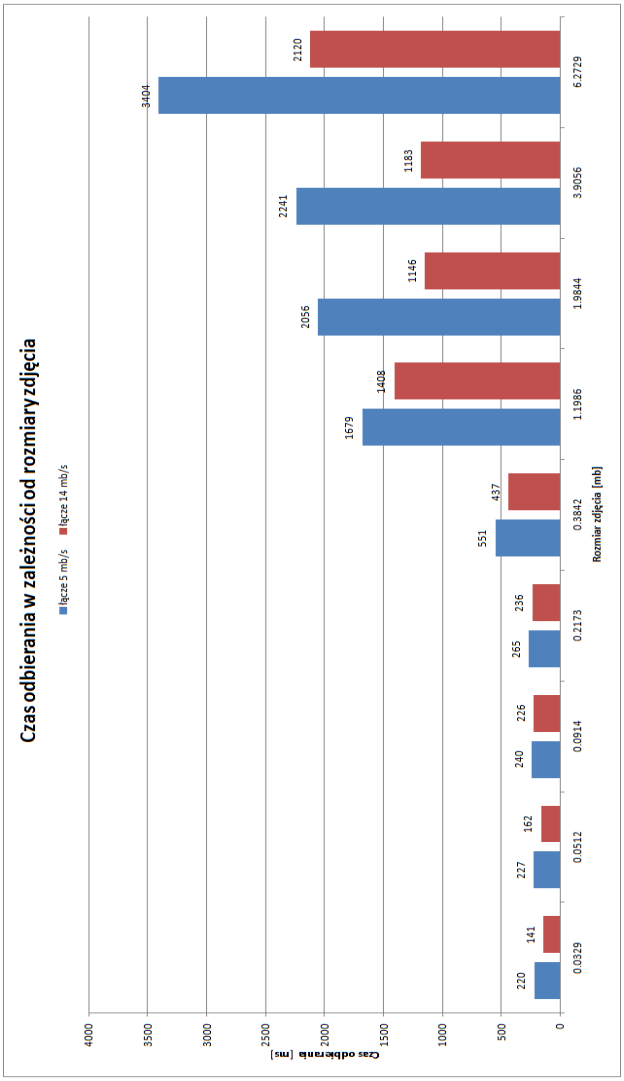
WYNIKI

Działanie aplikacji zostało zbadane pod wieloma aspektami i dokonano odpowiednich pomiarów. Na Rys. 6 przedstawiony został wykres czasu przesyłania zdjęcia do chmury w zależności od jego rozmiaru.



Rys. 6. Czas wysyłania zdjęć
(źródło: opracowanie własne)

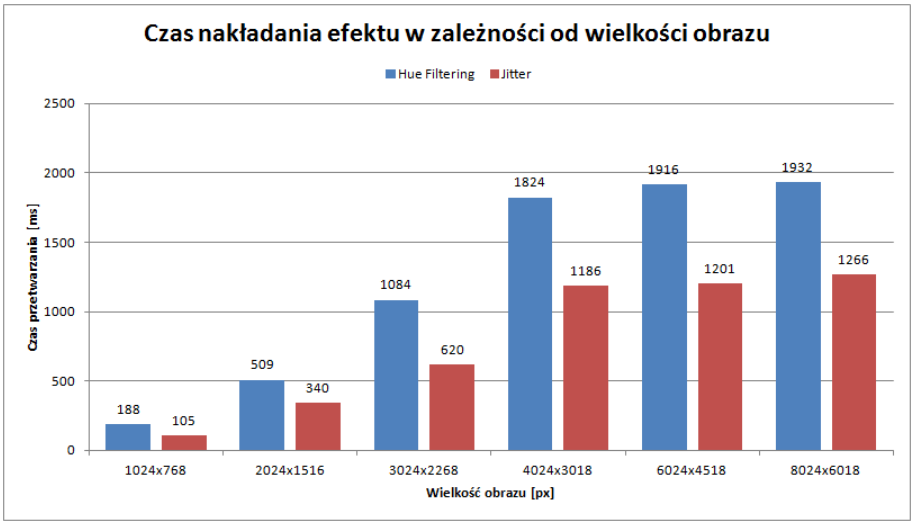
Zbadano dwie szybkości łącza. Pomimo, iż przedstawione wyniki są średnią kilku pomiarów nie otrzymano oczekiwanej zależności liniowej. Taki wynik spowodowany jest niedoskonałością łącza, które w krótkich przedziałach czasowych potrafi znacząco zmienić swoją przepustowość w przypadku znacznego obciążenia. Kolejny z wykresów przedstawiony na Rys. 7 obrazuje czas transferu w druga stronę, czyli z serwera do telefonu.



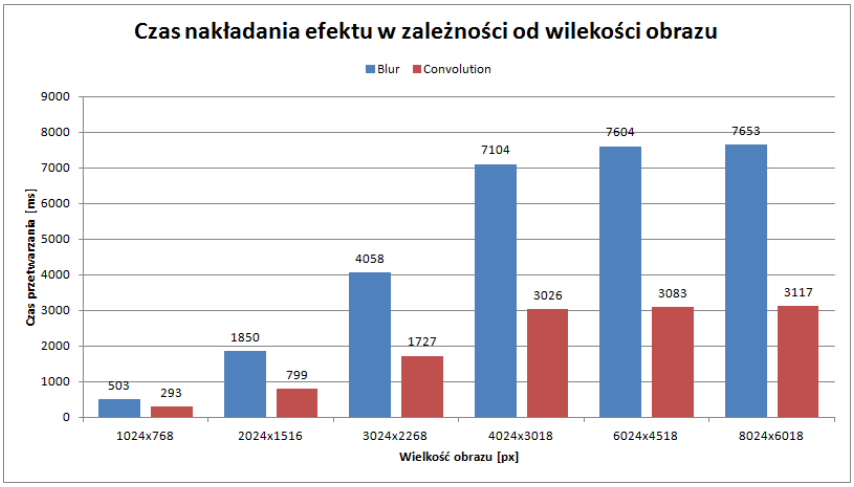
Rys. 7. Czas odbierania zdjęć
(źródło: opracowanie własne)

Z uwagi na to, że zazwyczaj prędkość downloadu znacząco przewyższa szybkość uploadu serie na Rys. 7 obrazują odpowiednio wyniki dla łącza o szybkości 5 oraz 14 Mb/s.
Wyraźnie widać, że nawet dla wolniejszego z badanych łącz czas pobierania obrazu o rozmiarze 6 Mb nie przekracza 4 sekund. W tym przypadku również nie otrzymano zależności liniowej.

Samo zestawienie czasu potrzebnego na przesłanie obrazu do i z serwera nie pozwala na ocenę wydajności napisanej aplikacji. Przesłanie obrazu w oryginalnym rozmiarze odbywa się jedynie na początku i końcu sesji obróbki zdjęcia. Pomiędzy tymi granicznymi momentami do klienta odsyłana jest pomniejszona wersja zdjęcia mimo, iż chmura ciągle pracuje nad oryginalnej wielkości plikiem. O tym, jak wydajne jest przetwarzanie obrazów w chmurze, pozwalają stwierdzić wykresy przedstawione na rysunkach 8, 9 oraz 10. Razem na Rys. 8 i 9 przedstawione są czasy działania czterech wybranych efektów dla sześciu różnej wielkości obrazów. Na Rys. 8 czas wykonania efektów nie przekracza 2 sekund. Natomiast na Rys. 9 zaprezentowano wyniki dla efektów bardziej czasochłonnych. Czas maksymalny czyli ok. 7,5 sekundy dla dość dużego obrazu jest wynikiem bardzo zadowalającym.



Rys. 8. Czas wykonania efektów Hue Filtering i Jitter
(źródło: opracowanie własne)



Rys. 9. Czas wykonania efektów Blur i Convolution
(źródło: opracowanie własne)

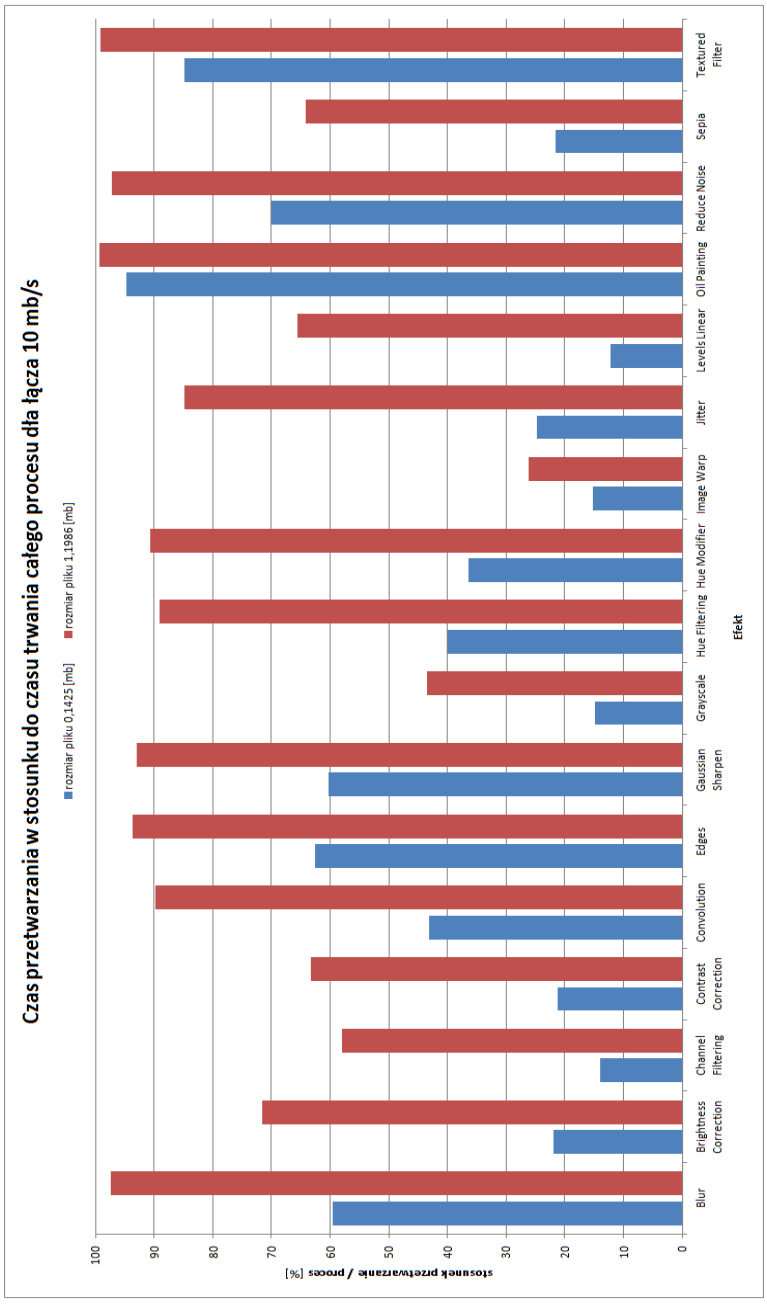
Na Rys. 10 przedstawione jest jak duży procent całego czasu oczekiwania na wynik wybranego efektu pochłania samo przetwarzanie pliku w chmurze obliczeniowej dla łącza o szybkości 10Mb/s. Zestawienie to obejmuje wszystkie dostępne w aplikacji efekty.

Dość wysoki udział procentowy czasu przetwarzania w chmurze do czasu całego procesu pozwala stwierdzić, że koszt związany z przeniesieniem logiki przetwarzania do chmury, jaki ponosi użytkownik aplikacji, jest stosunkowo niewielki w porównaniu do czasu, jaki tego typu przekształcenia musiałyby trwać na znacznie słabszym wydajnościowo urządzeniu mobilnym.

PODSUMOWANIE

Wykonana aplikacja prezentuje możliwości wykorzystania chmury jako zasobu do przetwarzania grafiki. Dobór narzędzi i technologii pozwolił na obróbkę zdjęć w chmurze sterowanej urządzeniem mobilnym. Dzięki wdrożeniu wytycznych Modern UI, uzyskano przejrzysty i wygodny interfejs do złudzenia przypominający pod względem wyglądu oraz obsługi domyślne aplikacje systemowe dla Windows Phone 8.

Stabilność i szybkość działania aplikacji przedstawiają się dość dobrze. Konieczny jest tylko dostęp do Internetu. Możliwości chmury są więc bardzo szerokie i przydatne nie tylko użytkownikom ściśle związanym z branżą IT, ale także w życiu codziennym.



Rys. 10. Czas przetwarzania efektu do czasu działania aplikacji (źródło: opracowanie własne)

LITERATURA

[1] Antonopoulos N., Gillam L.: *Cloud Computing: Principles, Systems and Applications*. Springer, 2010.

[1] Benmessaoud N., Williams C.J., Mudigonda U. M.: *Network Virtualization and Cloud Computing*. Microsoft Press, 2014.

[2] Erl T., Puttini R., Mahmood Z.: *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall, 2013.

[3] Dysart J.: *Cloud-based e-discovery can mean big savings for smaller firms*. ABA Journal, 2014.

[4] Smith D.M. et al.: *Predicts 2014: Cloud Computing Affects All Aspects of IT*. Gartner, 2014.

[5] Sosinsky B.: *Cloud Computing Bible*. Wiley Publishing , 2011.

[6] http://cloud-hosting-review.toptenreviews.com (dostęp 30.04.14).

[7] http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf (dostęp 30.04.14).

[8] http://www.ijstr.org/final-print/july2012/A-Review-Of-Cloud-Computing-Open-Architecture-And-Its-Security-Issues.pdf (dostęp 30.04.14).

[9] http://junereycasuga.github.io/blog/2013/10/12/cloud-computing-services-saas (dostęp 30.04.14).

[10] http://kainos.pl/blog/soa-–-service-oriented-architecture (dostęp 28.04.14).

[11] http://tarendo.pl/artykuly/jakie-sa-rodzaje-chmur-obliczeniowych (dostęp 28.04.14).

[12] http://www.salesforce.com/uk/socialsuccess/cloud-computing/why-move-to-cloud-10-benefits-cloud-computing.jsp (dostęp 15.04.14).

PAWEŁ HAWRYLAK[1], MACIEJ PAŃCZYK[2]

PORÓWNANIE NARZĘDZI DO DEBUGOWANIA PROGRAMÓW W JĘZYKACH C/C++ W ŚRODOWISKU LINUX

WSTĘP

Tworzenie oprogramowania nie jest procesem łatwym, a samo pisanie kodu programu jest jedynie jednym z jego etapów. Po napisaniu programu konieczne jest jego gruntowne przetestowanie. Niestety, popełnianie błędów jest immanentną cechą ludzką, dlatego mało który program jest pozbawiony błędów. Dotyczy to zwłaszcza dużych projektów. Szacuje się, że na 100 linii świeżo napisanego kodu programu przypada średnio około 2 błędów [13]. Mogą one prowadzić do nieoczekiwanego zachowania programu, marnowania zasobów sprzętowych i luk bezpieczeństwa, które mogą zostać wykorzystane przez atakującego. Wyszukiwanie, identyfikowanie i usuwanie błędów jest bardzo ważnym etapem tworzenia oprogramowania i często jest bardzo czasochłonne. Im wcześniej błędy zostaną usunięte, tym lepiej. Znalezienie poważnego błędu w programie przez wielu jego użytkowników kosztuje o wiele więcej niż znalezienie tego samego błędu przed wydaniem programu [17]. Z tego też powodu każdy programista powinien posiadać umiejętność lokalizowania błędów i korzystania z przeznaczonych do tego narzędzi.

Proces wyszukiwania i usuwania błędów w programie nosi nazwę debugowania (z ang. *debugging* – odrobaczanie, odpluskwianie). Etymologia pojęcia „bug" (z ang. robak, pluskwa) w kontekście błędu technicznego sięga czasów poprzedzających powstanie komputerów. Thomas Edison użył tego pojęcia w 1878 roku w jednym z listów do swojego współpracownika Theodore'a Puskasa [18]. Spopularyzowanie tego pojęcia w żargonie informatycznym powszechnie przypisuje się amerykańskiej pionierce informatyki Grace Hopper. Podczas prac nad komputerem Mark II prowadzonych w 1947 roku na Uniwersytecie Harvarda stwierdzono jego błędne działanie. Okazało się, że powodem

[1] Politechnika Lubelska, Wydział Elektrotechniki i Informatyki,
student kierunku Informatyka
[2] Politechnika Lubelska, Wydział Elektrotechniki i Informatyki, Instytut Informatyki

była ćma znajdująca się w jednym z przekaźników. Została ona usunięta i wklejona do dziennika wraz z podpisem „First actual case of bug being found", a naprawa tej usterki została określona jako debugowanie. Od tego momentu pojęcia „bug" i „debugowanie" są powszechnie używane w informatyce [15].

Na rynku dostępnych jest wiele różnych narzędzi wspomagających proces debugowania, które noszą nazwę debuggerów. Są one bardzo przydatne, ponieważ wyszukiwanie błędów poprzez analizę samego kodu programu lub dodanie do niego dodatkowego kodu śledzącego jest niekiedy bardzo uciążliwe i czasochłonne. Najbardziej znanym debuggerem dla systemu GNU/Linux jest GNU Debugger, który umożliwia krokowe wykonanie programu i dynamiczną jego analizę. Możliwe jest dzięki temu sprawdzenie wartości rejestrów i zmiennych w trakcie wykonania programu. W przypadku języków C i C++ bardzo istotną rolę odgrywają błędy związane z zarządzaniem pamięcią. Wynika to z faktu, iż za zarządzanie pamięcią odpowiedzialny jest programista, a nie środowisko uruchomieniowe i odśmiecacz pamięci (ang. *garbage collector*) jak np. w języku Java. Z tego powodu powstało bardzo wiele narzędzi, które stworzone zostały w celu wykrywania tylko tego rodzaju błędów. Takie mechanizmy są obecne również w bibliotece GNU C i kompilatorach GCC. Użycie debuggerów pamięci jest zazwyczaj stosunkowo proste, ponieważ błędy są wykrywane i zgłaszane przez debugger (często wraz z lokalizacją) [3].

W niniejszej pracy przedstawione i porównane zostaną najpopularniejsze narzędzia do debugowania programów w językach C i C++ dla systemu GNU/Linux, a ich praktyczne użycie zostanie zaprezentowane na własnych [2] przykładach błędów programistycznych.

NAJCZĘSTSZE BŁĘDY POPEŁNIANE PRZY PROGRAMOWANIU W JĘZYKACH C I C++

Poznanie specyfiki typowych błędów programistycznych jest konieczne do pisania poprawnych i szybkich programów, a także do skutecznego ich debugowania. W tym rozdziale przedstawione zostaną dwa najbardziej typowe błędy popełniane przy programowaniu w językach C i C++, którymi są wycieki pamięci i przepełnienia bufora.

WYCIEK PAMIĘCI

Wyciek pamięci (ang. *memory leak*) to błąd polegający na nie zwolnieniu zarezerwowanej pamięci, gdy nie jest już potrzebna w programie. Języki C i C++ nie posiadają „odśmiecacza" pamięci (ang. *garbage collector*), dlatego cała odpowiedzialność za rezerwację i zwalnianie pamięci dynamicznej na stercie spoczywa na programiście. Duża ilość wycieków powoduje wzrastające z czasem zużycie pamięci, co w najgorszym przypadku może doprowadzić do jej wyczerpania. Najbardziej dotkliwe są one dla aplikacji działających nieprzerwanie przez dłuższy czas, ale należy się ich wystrzegać w każdym programie [9].

Na listingu 1 przedstawiono przykładowy program z wyciekiem pamięci w języku C.

Listing 1. Przykładowy program z wyciekiem pamięci w języku C

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  /* srednia: Funkcja zwraca średnią z dwóch liczb */
 5  double srednia(double liczba1, double liczba2)
 6  {
 7    double *temp; //deklaracja wskaźnika
 8    temp = malloc(sizeof *temp); //alokacja pamięci
 9    if (temp == NULL) //sprawdzenie poprawności alokacji
10    {
11      exit(EXIT_FAILURE); //wyjście z programu
12    }
13    else
14    {
15      *temp = (liczba1 + liczba2) / 2.0; //obliczenie
    średniej
16      return *temp; //zwrócenie wartości wynikowej
17    }
18  }
19  int main(void)
20  {
21    int i = 0;
22    while(1)
23    {
24      printf("Srednia z %d i 10 to
25  %f\n",i,srednia(i,10));
        i++;
26    }
27    return EXIT_SUCCESS;
28  }
```

Przedstawiony program oblicza i wyświetla średnie arytmetyczne z liczby 10 i kolejnych liczb naturalnych w pętli nieskończonej. Funkcja *srednia()* przyjmuje jako argumenty dwie liczby i oblicza średnią przy pomocy zmiennej dynamicznej utworzonej z wykorzystaniem funkcji *malloc()*. Można zauważyć, że wynik zwracany jest przez wartość, a pamięć wcześniej przydzielona nie jest zwalniana po wykorzystaniu za pomocą funkcji *free()*. Wynikiem tego jest pozostawienie zarezerwowanego obszaru pamięci bez możliwości jego dalszego wykorzystania w programie, ponieważ wskaźnik na ten obszar zostaje utracony wraz z wyjściem z funkcji. Każde kolejne wywołanie tej funkcji w pętli zawartej w głównym programie powoduje rezerwację kolejnego obszaru pamięci bez jego zwalniania. Z tego powodu program z biegiem czasu zajmuje coraz więcej pamięci do momentu aż wykorzystana zostanie cała dostępna pamięć lub użytkownik zakończy działanie programu.

Na listingu 2 przedstawiono podobny przykład w języku C++.

Listing 2. Przykładowy program z wyciekiem pamięci w języku C++

```
 1  #include <iostream>
 2  #include <cstdlib>
 3
 4  using namespace std;
 5
 6  /* losujPodzielna: Funkcja losuje liczby z zakresu
 7    0-999 i zwraca pierwszą liczbę podzielną przez 17 */
 8  int losujPodzielna()
 9  {
10    int *wylosowana = new int; //deklaracja i alokacja
11    while (true)
12    {
13      *wylosowana = rand() % 1000; //losowanie
14      if (!(*wylosowana % 17))//sprawdzenie podzielności
15        return *wylosowana; // zwrócenie wyniku
16    }
17  }
18
19  int main()
20  {
21    srand(time(NULL));
22    while (true)
23    {
24      cout << losujPodzielna() << "\n";
25    }
26  }
```

Przedstawiony program losuje w pętli nieskończonej liczby z zakresu 0-999 i wyświetla te podzielne przez 17. Losowanie liczb i zwracanie pierwszej spełniającej te kryteria ma miejsce w funkcji *losujPodzielna()*. W funkcji tej wykorzystywana jest zmienna dynamiczna utworzona z wykorzystaniem operatora *new* używanego do alokacji pamięci w języku C++. Sytuacja jest analogiczna do poprzedniej – funkcja zwraca wynik przez wartość, a niepotrzebny już obszar pamięci nie jest zwalniany po wykorzystaniu za pomocą operatora właściwego w tym przypadku dla C++, czyli *delete*. Ma to takie same konsekwencje jak poprzednio – program podczas działania zajmuje coraz więcej pamięci.

Poprawione programy przedstawiono na listingach 3 i 4.

Listing 3. Poprawiony program w języku C z listingu 1

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  /* srednia: Funkcja zwraca średnią z dwóch liczb */
 5  double* srednia(double liczba1, double liczba2)
 6  {
 7    double *temp; //deklaracja wskaźnika
 8    temp = malloc(sizeof *temp); //alokacja pamięci
 9    if (temp == NULL) //sprawdzenie poprawności alokacji
10    {
11      exit(EXIT_FAILURE); // wyjście z programu
12    }
13    else
14    {
15      *temp=(liczba1 + liczba2)/2.;//obliczenie średniej
16      return temp; // zwrócenie wskaźnika
17    }
18  }
19  int main(void)
20  {
21    int i = 0;
22    while(1)
23    {
24      double *wynik = srednia(i,10);
25      printf("Srednia z %d i 10 to %f\n", i, *wynik);
26      free(wynik); // zwolnienie pamięci
27      wynik = 0;
28      i++;
29    }
30    return EXIT_SUCCESS;
31  }
```

Dzięki zmianie typu zmiennej zwracanej przez funkcję *srednia()* z double na wskaźnik na double, możliwe stało się zwolnienie zarezerwowanej pamięci w programie głównym. Funkcja *srednia()* alokuje pamięć, umieszcza w niej wynik i zwraca wskaźnik na tą pamięć. W pętli zawartej w programie głównym adres przez nią zwrócony jest przypisywany do wskaźnika i wykorzystywany do wyświetlenia wyniku, po czym pamięć jest zwalniana za pomocą funkcji *free()*.

Następnie do wskaźnika przypisywana jest wartość 0, ponieważ pamięć, na którą wskazuje, została już zwolniona (nie jest w tym przypadku konieczne, ale warto to robić po każdym zwolnieniu pamięci, aby uniknąć nieprzewidywalnych błędów występujących przy próbie dostępu do zwolnionego obszaru pamięci).

Na listingu 4 przedstawiono poprawiony program w języku C++ z listingu 2.

Listing 4. Poprawiony program w języku C++ z listingu 2

```
 1 #include <iostream>
 2 #include <cstdlib>
 3
 4 using namespace std;
 5
 6 /* losujPodzielna: Funkcja losuje liczby z zakresu
 7 0-999 i zwraca pierwszą liczbę podzielną przez 17 */
 8 void losujPodzielna(int *wylosowana)
 9 {
10   while (true)
11   {
12     *wylosowana = rand() % 1000; //losowanie
13     if (!(*wylosowana % 17))//sprawdzenie podzielności
14       return; // powrot z funkcji
15   }
16 }
17 int main()
18 {
19   srand(time(NULL));
20   while (true)
21   {
22     int *wylosowana = new int; //deklaracja i alokacja
23     losujPodzielna(wylosowana);
24     cout << *wylosowana << "\n";
25     delete wylosowana; // zwolnienie pamieci
26     wylosowana = 0;
27   }
28 }
```

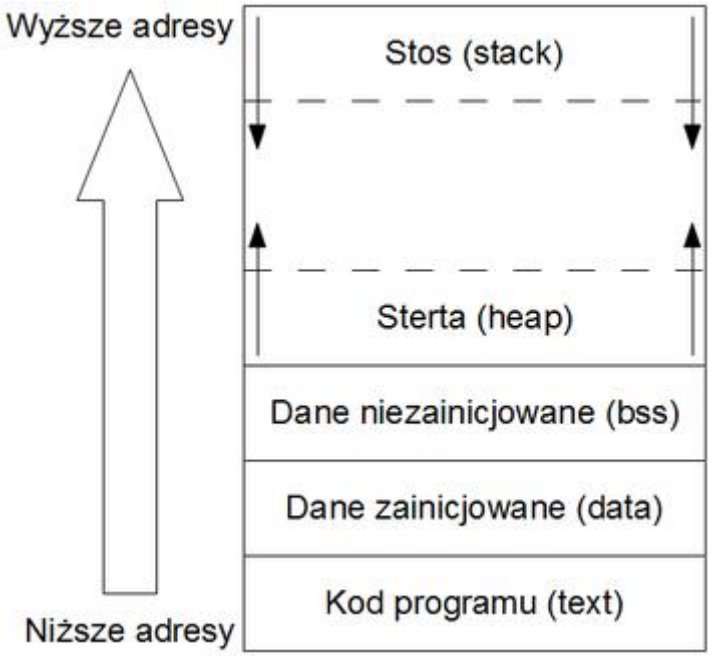
W tym przypadku wyeliminowano wyciek pamięci w trochę inny sposób. Funkcja *losujPodzielna()* nie zwraca już wyniku bezpośrednio. Wykorzystuje w tym celu obszar pamięci wskazywany przez wskaźnik przekazany w argumencie funkcji. Pamięć jest alokowana (*new*) i zwalniana (*delete*) w programie głównym. Podobnie jak poprzednio wskaźnik jest zerowany po zwolnieniu.

Powyższe zmiany spowodowały, że wykorzystanie pamięci przez programy pozostaje mniej więcej stałe przez cały czas ich działania. Najprostszym rozwiązaniem tego problemu byłoby napisanie powyższych programów z wykorzystaniem zwykłych zmiennych. Rzadko jednak da się to zrobić w przypadku bardziej rozbudowanych programów. Należy w związku z tym zawsze pamiętać o tym, by każdej wykonanej instrukcji alokacji pamięci towarzyszyła odpowiednia instrukcja jej zwalniania, czyli *free()* (w C) lub *delete/delete[ ]* (w C++) [11]. W programach posiadających wiele funkcji czy klas współdziałających ze sobą nie jest to wbrew pozorom zadanie trywialne, co jest jednym z powodów powstania języków wykorzystujących mechanizm odśmiecania pamięci.

PRZEPEŁNIENIE BUFORA

Przepełnienie bufora (ang. buffer overflow) to błąd polegający na zapisie danych poza granicami zarezerwowanego obszaru pamięci (bufora). Powoduje to nadpisanie danych sąsiadujących z buforem. Jest to poważny problem, ponieważ mogą w ten sposób zostać nadpisane istotne dla programu dane. Konsekwencją tego może być zwracanie niepoprawnych wyników przez program, naruszenie ochrony pamięci, a także zmiana logiki wykonania programu w zależności od tego, jakie dane zostaną nadpisane. Pewną trudnością jest to, iż błędy te nie zawsze występują zaraz po przepełnieniu bufora i mogą się ujawnić dopiero w dalszych częściach programu [9]. Błąd przepełnienia bufora często może być wykorzystany przez atakującego do ominięcia zabezpieczeń programu lub wykonania złośliwego kodu, czego efektem może być przejęcie kontroli nad atakowanym systemem [8].

Przyczyną przepełnień bufora jest brak kontroli rozmiaru danych wprowadzanych do bufora. Najczęściej dotyczy to tablic znaków. Języki C i C++ nie posiadają żadnych wewnętrznych mechanizmów zabezpieczających, które sprawdzałyby, czy dane zapisywane do bufora się w nim mieszczą. To programista piszący kod musi brać pod uwagę możliwość wystąpienia takich błędów i starać się nie dopuścić do takiej sytuacji [1]. Błędy przepełnienia bufora można podzielić ze względu na obszar pamięci w jakim występują. Na rysunku 1 przedstawiono przestrzeń adresową typowego procesu.



Rysunek 1. Przestrzeń adresowa procesu
(źródło: opracowanie własne na podstawie [1])

Jak widać, przestrzeń adresowa procesu podzielona jest na kilka segmentów. Segment text jest obszarem pamięci, który zawiera kod wykonywalny programu i służy tylko do odczytu. Segmenty data i BSS (Block Started by Symbol) można określić ogólnie jako pamięć statyczną. Segment data zawiera zmienne globalne i statyczne zainicjowane w sposób jawny przez programistę do wartości różnych od 0. Segment BSS zawiera z kolei zmienne globalne i statyczne niezainicjowane w sposób jawny (zainicjowane automatycznie do 0) lub zainicjowane przez programistę do 0. Sterta jest obszarem pamięci dynamicznej obsługiwanym za pomocą funkcji takich jak *malloc(), calloc(), realloc(), free()* (C) oraz operatorów *new* i *delete* (C++). Obszar ten wykorzystywany jest również przez biblioteki współdzielone. Rozmiar sterty nie jest stały i w razie potrzeby jest on dostosowywany poprzez wywołania systemowe. Segment stosu jest zorganizowany jako struktura typu LIFO (Last In, First Out – ostatni na wejściu, pierwszy na wyjściu). Adres wierzchołka stosu przechowywany jest w specjalnym rejestrze procesora (RSP w przypadku architektury x86-64), a do jego obsługi wy-

korzystywane są instrukcje procesora *push* (odłóż) i *pop* (zdejmij). Na stosie odkładane są dane kontekstowe związane z wywołaniami funkcji takie jak parametry funkcji, adres powrotu z funkcji, zachowany wskaźnik ramki stosu i zmienne lokalne. Można zauważyć, że sterta i stos rosną w przeciwnych kierunkach, zbliżając się do siebie. Dzięki temu pamięć jest optymalnie wykorzystana [1].

Gdy znana jest już struktura pamięci w programie, można przystąpić do zaprezentowania różnych rodzajów przepełnień bufora. Na listingu 5 przedstawiono przykładowy program z przepełnieniem bufora w obszarze pamięci statycznej (konkretnie data).

Listing 5. Przykład przepełnienia bufora w obszarze danych statycznych w języku. C

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 char poprzednia[7] = "00000";
 6 char globalna[7] = "00000";
 7 char nastepna[7] = "00000";
 8
 9 int main(void)
10 {
11   strcpy(globalna, "ABCDEF");
12   printf("Przepełnienie o 1 bajt: %s\n", globalna);
13   strcpy(globalna, "ABCDEFGHIJ");
14   printf("Przepełnienie o 5 bajtów: %s\n", globalna);
15   *(globalna-1) = 'X';
16   printf("Zamazanie 1 bajta danych przed: %s\n",

17 globalna);
18   printf("\nPoprzednia zmienna: %s\n", poprzednia);
19   printf("Następna zmienna: %s\n", nastepna);
20
21   return 0;
   }
```

Zaprezentowany przykład wykorzystuje 3 zainicjowane globalne tablice znaków o rozmiarze 6 bajtów. Do drugiej z tablic kopiowane są kolejno dwa ciągi znaków – jeden przekraczający rozmiar tablicy o 1 bajt, drugi o 5 bajtów.
Ponadto, bezpośrednio przed tablicą zapisywany jest jeden bajt danych. Na listingu 6 przedstawiono wynik działania programu.

Listing 6. Efekt działania programu z listingu 5

```
[p@arch src]$ gcc -Wall -o przyklad3a przyklad3a.c &&
./przyklad3a
Przepełnienie o 1 bajt: ABCDEF
Przepełnienie o 5 bajtów: ABCDEFGHIJ
Zamazanie 1 bajta danych przed: ABCDEFGHIJ

Poprzednia zmienna: 00000XABCDEFGHIJ
Następna zmienna: GHIJ
```

Jak widać, przekroczenie granic tablicy globalna powoduje nadpisanie komórek pamięci należących do sąsiednich tablic. W związku z tym, że w językach C i C++ łańcuchy znaków kończą się bajtem o wartości 0 i na tej podstawie określana jest ich długość, nadpisanie go prowadzi do wyświetlania dalszych komórek pamięci aż do momentu napotkania bajtu o tej wartości.

Najbardziej typowe błędy przepełnienia bufora dotyczą buforów umieszczonych na stercie i stosie. To właśnie tego typu błędy występują najczęściej i są powszechnie wykorzystywane przez atakujących. Jak wspomniano wcześniej, na stercie zachodzi dynamiczna alokacja pamięci i najczęściej przechowywane są na niej dane programu. Na listingu 7 przedstawiono przykładowy program z przepełnieniem bufora na stercie.

Listing 7. Przykład przepełnienia bufora na stercie w języku C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv)
{
  char *bufor;
  int *liczba_znakow;

  bufor = malloc(6 * sizeof(*bufor));
  liczba_znakow = malloc(sizeof(*liczba_znakow));

  *liczba_znakow = strlen(argv[1]);
  strcpy(bufor, argv[1]);
  *(bufor-1) = 'X';

  printf("Bufor: %s\n", bufor);
  printf("Liczba znakow: %d\n", *liczba_znakow);

```

```
20    free(liczba_znakow);
21    free(bufor);
22
23    return 0;
24  }
```

Działanie tego programu jest bardzo proste. Wykorzystuje on dwie zmienne alokowane dynamicznie – tablicę znaków o rozmiarze 6 bajtów oraz zmienną typu *int*. Program przyjmuje w parametrze łańcuch znaków, którego liczba znaków jest zapisywana w zmiennej typu *int*, a następnie sam ciąg jest zapisywany do bufora. Przekazanie do programu zbyt dużego ciągu znaków spowoduje przepełnienie bufora, a jeśli przepełnienie będzie odpowiednio duże, to dokonana zostanie modyfikacja zmiennej przechowującej liczbę znaków.

Dochodzi ponadto do zapisu jednego bajta danych przed buforem bezpośrednio w kodzie programu. Następnie wyświetlana jest zawartość bufora i zapisana liczba znaków, po czym następuje zwolnienie pamięci oraz zakończenie programu.

Na listingu 8 przedstawiono efekt działania tego programu.

Listing 8. Efekt działania programu z listingu 7

```
 1  [p@arch src]$ gcc -Wall -o przyklad3b przyklad3b.c
 2  [p@arch src]$ ./przyklad3b ABCDEF
 3  Bufor: ABCDEF
 4  Liczba znakow: 6
 5  [p@arch src]$ ./przyklad3b ABCDEFGHIJ
 6  Bufor: ABCDEFGHIJ
 7  Liczba znakow: 10
 8  [p@arch src]$ ./przyklad3b ABCDEFGHIJKLMNOPQRSTUVWXY-
    ZYXWVUTSRQ
 9  Bufor: ABCDEFGHIJKLMNOPQRSTUVWXYZYXWVUTSRQ
10  Liczba znakow: 5329491
```

Można zauważyć, że program mimo takiego samego zadeklarowanego rozmiaru bufora jak w przykładzie z listingu 5, działa poprawnie dla małych przepełnień. Wynika to z faktu, iż w rzeczywistości alokowany jest większy obszar pamięci niż zadeklarowany. Alokator wyrównuje bowiem dane w pamięci i przechowuje dodatkowe dane dotyczące danego przydziału takie jak np. rozmiar (w celu jego późniejszego zwolnienia).

W bibliotece GNU C bloki pamięci na stercie zarządzane są z wykorzystaniem metody ”boundary tags”. Metoda ta w ogólności polega na tym, że bloki pamięci posiadają pola graniczne, w których przechowywane są informacje dotyczące rozmiaru danego obszaru i jego statusu. W momencie zwolnienia zarezerwowanego bloku pamięci sprawdzany jest nagłówek następnego bloku pamięci i stopka poprzedniego. Jeśli sąsiadujący blok nie jest w użyciu, to są one łączone w jeden w celu uformowania większego wolnego bloku. Umożliwia to szybką konsolidację pofragmentowanych wolnych bloków pamięci. Wolne bloki są przechowywane w postaci list dwukierunkowych [22].

W przypadku rozważanej architektury x86-64 i biblioteki GNU C domyślne wyrównanie wynosi 16 bajtów *(2 * sizeof(size t))*, a na zarezerwowany blok pamięci przypada co najmniej 8 bajtów danych alokatora. Minimalny rozmiar alokacji wynosi natomiast 32 bajty. W związku z tym, mimo że w programie chciano zarezerwować 6 bajtów pamięci, alokator przydzielił tak naprawdę obszar 32-bajtowy (podobnie w przypadku zmiennej *liczba_znakow*). Oznacza to, że aby nadpisać zmienną *liczba_znakow* trzeba było zapisać do bufora co najmniej 33 bajty. Należy przy tym pamiętać, że do nadpisania danych alokatora wystarczy mniej, bo 25 bajtów. Nadpisanie tych danych może doprowadzić do zatrzymania programu i wyświetlenia błędu, jeśli włączone są wbudowane mechanizmy sprawdzania spójności sterty biblioteki GNU C [12].

Najbardziej niebezpieczne są przepełnienia bufora mające miejsce na stosie. Podczas wywołania funkcji na stos odkładane są parametry wywołania, adres powrotu, wskaźnik ramki i zmienne lokalne danej funkcji. W ten sposób powstaje ramka stosu, czyli fragment stosu przynależący do danego wywołania funkcji. W momencie wyjścia z funkcji ramka stosu jest usuwana, a wykonanie programu wraca do miejsca wskazywanego przez adres powrotu.

W związku z tym, że na rozpatrywanej architekturze stos rośnie w kierunku zera, w prosty sposób możliwe staje się nadpisanie adresu powrotu poprzez przepełnienie bufora lokalnego utworzonego na stosie. Pozwala to na zmianę przepływu wykonania programu i wykonanie złośliwego kodu wprowadzonego do bufora np. w celu eskalacji uprawnień i przejęcia kontroli nad systemem [1].

Na listingu 9 przedstawiono przykładowy program zawierający przepełnienie bufora w obszarze stosu. Przedstawiony program jest bardzo prosty, ale pokazuje istotę sprawy. Do lokalnej tablicy znaków o rozmiarze 6 bajtów umieszczonej na stosie kopiowany jest łańcuch znaków przekazany w parametrze do programu. Następnie zawartość bufora jest wyświetlana. Efekt działania programu został przedstawiony na listingu 10.

Listing 9. Przykład przepełnienia bufora na stosie w języku C++

```
 1  #include <iostream>
 2  #include <cstring>
 3
 4  using namespace std;
 5
 6  int main(int argc, char **argv)
 7  {
 8    char bufor[7];
 9
10    strcpy(bufor, argv[1]);
11
12    cout << bufor << '\n';
13  }
```

Listing 10. Efekt działania programu z listingu 9

```
 1  [p@arch src]$ gcc -Wall -o przyklad3b przyklad3b.c
 2  [p@arch src]$ ./przyklad3b ABCDEF
 3  Bufor: ABCDEF
 4  Liczba znakow: 6
 5  [p@arch src]$ ./przyklad3b ABCDEFGHIJ
 6  Bufor: ABCDEFGHIJ
 7  Liczba znakow: 10
 8  [p@arch src]$ ./przyklad3b ABCDEFGHIJKLMNOPQRSTUVWXY-
    ZYXWVUTSRQ
 9  Bufor: ABCDEFGHIJKLMNOPQRSTUVWXYZYXWVUTSRQ
10  Liczba znakow: 5329491
```

Jak widać, dopiero zapisanie w buforze 25 bajtów spowodowało błąd programu. Wynika to z faktu, iż wyrównanie danych na stosie w przypadku architektury x86-64 wynosi 16 bajtów. Bufor zajął więc w rzeczywistości 16 bajtów, a nie 6. Bezpośrednio za buforem znajdował się zachowany wskaźnik ramki o rozmiarze 8 bajtów, którego nadpisanie w tym przypadku nie spowodowało żadnych nieprawidłowości (program zakończył się zaraz po wyjściu z funkcji *main()*, więc błędna wartość nawet nie została użyta). Zaraz obok niego znajdował się 8-bajtowy adres powrotu [14]. Zapisanie 25 bajtów spowodowało więc nadpisanie części obszaru należącego do adresu powrotu. Wyjście z funkcji *main()* spowodowało skok w niewłaściwe miejsce wskazywane przez adres powrotu i zakończyło się błędem naruszenia ochrony pamięci, ponieważ proces nie miał dostępu do tego obszaru.

DEBUGOWANIE ZA POMOCĄ NARZĘDZI GNU

W niniejszym rozdziale przedstawione zostaną narzędzia do debugowania powiązane z projektem GNU. Będą to GNU Debugger, mechanizmy śledzenia pamięci i kontroli spójności sterty w bibliotece GNU C oraz mechanizm ochrony stosu w kompilatorach GCC (GNU Compiler Collection).

GNU DEBUGGER

GNU Debugger (znany też pod nazwą GDB) jest programem śledzącym, będącym częścią projektu GNU. Został napisany w 1986 roku przez Richarda Stallmana, jednego z najbardziej rozpoznawalnych twórców ruchu wolnego oprogramowania i założyciela Free Software Foundation (FSF) i projektu GNU. Jest on również twórcą kompilatora GCC i edytora Emacs. GDB oparte jest na licencji GNU General Public License (GPL). Obecnie rozwijane jest przez grupę programistów wyznaczonych przez FSF. W momencie pisania tej pracy najnowszą wersją GDB jest wersja 7.5.1 datowana na dzień 29 listopada 2012 roku.

GDB dostępne jest na wiele architektur sprzętowych i działa na wielu systemach operacyjnych (uniksopodobne i Windows). Wspiera też w różnym stopniu wiele języków programowania (Ada, C, C++, D, Fortran, Go, Java, Modula-2, Objective-C, OpenCL C, Pascal), a nie wspierane mogą być debugowane z wykorzystaniem pseudojęzyka nazwanego minimal.

Debugger ten umożliwia (za: [5]):

- kontrolowane wykonanie programu instrukcja po instrukcji (danego języka programowania lub assemblera);
- zatrzymanie programu w ustalonych warunkach (np. linia programu, wartość zmiennej);
- sprawdzenie zawartości pamięci i wartości zmiennych w trakcie debugowania;
- modyfikację danych i działania programu podczas jego wykonania;
- cofanie wykonania programu.

GDB działa w trybie tekstowego wiersza poleceń, jednak istnieją liczne nakładki (front-end) dodające do niego graficzny interfejs użytkownika.
Do najpopularniejszych należą Data Display Debugger (DDD), KDbg, Nemiver, xxgdb, czy też Insight.

Ponadto wiele zintegrowanych środowisk programistycznych (IDE) oferuje wsparcie dla GDB, np. Qt Creator, Eclipse, KDevelop, NetBeans, Code::Blocks i Codelite. W pracy tej nie zostaną one jednak przedstawione, ponieważ nie są one potrzebne do procesu debugowania i nie zmieniają w szczególny sposób jego właściwości.

Aby debugować program za pomocą GDB konieczne jest dołączenie do jego pliku wykonywalnego informacji dla debuggera m.in. na temat zmiennych, typów, stałych i funkcji, które w procesie kompilacji są tracone. W przypadku programów pisanych w językach C i C++ i kompilowanych za pomocą kompilatorów GCC można to zrobić za pomocą przełącznika *-g* dodanego do polecenia kompilacji [4]. Dobrze jest też jawnie wyłączyć optymalizacje kompilatora za pomocą opcji *-O0*, ponieważ w pewnych sytuacjach potrafią one znacznie skomplikować debugowanie programu. Ostateczne polecenie kompilacji może wyglądać np. tak:

gcc -O0 -g -o przyklad przyklad.c (dla C)

g++ -O0 -g -o przyklad przyklad.cpp (dla C++)

Debugowanie programu można rozpocząć na kilka sposobów. Najczęściej używanym jest uruchomienie programu *gdb* z parametrem w postaci ścieżki do programu.

Program można też załadować do GDB, podając ścieżkę w parametrach poleceń *file* lub *exec-file* po uruchomieniu debuggera. W przypadku drugiego polecenia należy pamiętać o załadowaniu tablicy symboli (informacji debugowania) za pomocą polecenia *symbol-file* z parametrem w postaci ścieżki do pliku. Dzięki poleceniu *attach* można przyłączyć debugger pod działający już proces, podając jego identyfikator PID w parametrze. Powoduje to zatrzymanie wykonania danego procesu.

Można tego również dokonać uruchamiając program *gdb*, podając w pierwszym parametrze ścieżkę do programu, a w drugim numer PID procesu. Od procesu można się odłączyć za pomocą polecenia *detach*. Do uruchomienia załadowanego programu służy polecenie *run*. W jego parametrach można podać parametry wywołania programu. Można to też zrobić ustawiając je wcześniej za pomocą polecenia *set args*.

GDB umożliwia również analizę zrzutów pamięci (ang. *core*) tworzonych przez system operacyjny w momencie błędnego zakończenia procesu. W tym celu uruchamia się go z dwoma parametrami – ścieżką do programu i do pliku zrzutu pamięci lub wykorzystuje się polecenia GDB: *file/exec-file* i *core* z odpowiednimi parametrami [5].

Listing 11. Przykładowa sesja debuggera GDB

```
 1  [p@arch src]$ gcc -O0 -g -o przyklad przyklad.c
 2  [p@arch src]$ gdb ./przyklad
 3  GNU gdb (GDB) 7.5.1
 4  Copyright (C) 2012 Free Software Foundation, Inc.
 5  License GPLv3+: GNU GPL version 3 or later
    <http://gnu.org/licenses/gpl.html>
 6  This is free software: you are free to change and

 7  redistribute it.
 8  There is NO WARRANTY, to the extent permitted by law.
    Type "show copying"
 9  and "show warranty" for details.
10  This GDB was configured as "x86_64-unknown-linux-gnu".
11  For bug reporting instructions, please see:
12  <http://www.gnu.org/software/gdb/bugs/>...
13  Reading symbols from /media/sf_src/przyklad...done.
14  (gdb) run
15  Starting program: /media/sf_src/przyklad
    warning: Could not load shared library symbols for
16
17  linux-vdso.so.1.
18  Do you need "set solib-search-path" or "set sysroot"?
19  [Inferior 1 (process 1053) exited normally]
    (gdb) quit
    [p@arch src]$
```

W linii 3. powyższej przykładowej sesji programu GDB widać, że wersja programu GDB to 7.5.1. W linijce 9. można zobaczyć na jakiej architekturze i systemie operacyjnym został uruchomiony. W tym przypadku jest to architektura x86-64 i system GNU/Linux. W linii 12. widać komunikat o pomyślnym załadowaniu symboli (informacji dla debuggera) z pliku wykonywalnego. Po uruchomieniu programu za pomocą polecenia *run* wyświetlone zostało ostrzeżenie, jednak nie jest ono istotne i można je zignorować. W linii 17. widać, że proces został zakończony normalnie (a więc funkcja *main()* zwróciła wartość równą 0). Następnie debugger został zamknięty za pomocą polecenia *quit*.

W celu zatrzymania programu w pewnym ustalonym momencie i analizy jego stanu lub wykonania go od tego miejsca krok po kroku należy przed użyciem polecenia *run* ustawić punkty przerwania (ang. breakpoints) za pomocą polecenia *break*. Punkt przerwania może być określony za pomocą nazwy funkcji, numeru linii aktualnego pliku źródłowego, przesunięcia o określoną liczbę wierszy w przód lub tył za pomocą +przesunięcie i -przesunięcie, pary nazwa pli-

ku:numer linii i nazwa pliku:nazwa funkcji lub adresu w pamięci w postaci **adres*, a każdemu z nich jest przypisywany indywidualny numer. Powoduje to zatrzymanie wykonania programu w określonym miejscu, przy czym można także określić dodatkowe warunki zatrzymania programu, pisząc na końcu polecenia if *warunek* lub używając polecenia *condition numer breakpointa warunek* dla istniejących punktów przerwania [3].

Warto zauważyć, że GDB rozpoznaje funkcje przeciążone np. w języku C++ i w momencie, gdy istnieje wiele funkcji o tej samej nazwie, pozwala wybrać wszystkie lub tylko niektóre z nich. Podobną rolę spełnia watchpoint ustawiany za pomocą polecenia *watch* i catchpoint ustawiany za pomocą polecenia *catch*. Są to specjalne punkty przerwania, z których pierwszy zatrzymuje program w momencie zmiany wartości wyrażenia niezależnie od miejsca, w którym to następuje, a drugi w momencie wystąpienia określonego zdarzenia, takiego jak np. wystąpienie wyjątku C++ lub ładowanie biblioteki.

Do usuwania punktów przerwania służy polecenie *clear* przyjmujące argumenty analogiczne do polecenia *break* oraz *delete* przyjmujące w parametrze numer przerwania. Wywołanie drugiego polecenia bez argumentu powoduje usunięcie wszystkich breakpointów. Punkty te można też wyłączać i włączać za pomocą poleceń *disable* i *enable*, które w parametrach przyjmują ich numery. Do wyświetlenia informacji o breakpointach służy polecenie *info break* [5].

Do krokowego wykonania programu po jego zatrzymaniu używane są polecenia *step* i *next* oraz *stepi* i *nexti*. Polecenie *step* powoduje wykonanie programu aż do momentu osiągnięcia nowego wiersza w kodzie źródłowym, przy czym kod wywoływanych funkcji także jest wykonywany krokowo (o ile posiadają one informacje dla debuggera). Polecenie *next* działa podobnie, ale nie wchodzi do kodu wywoływanych funkcji tylko wykonuje je całe.

Pozostałe dwa polecenia są analogiczne z tą różnicą, że dotyczą instrukcji języka maszynowego, a nie kodu źródłowego. Polecenia te można wykonywać z parametrem wskazującym liczbę ich powtórzeń. W celu kontynuowania wykonania programu aż do pewnego konkretnego punktu można użyć polecenia *until* z parametrem w postaci analogicznej do polecenia *break*. Wywołanie tego polecenia bez parametru powoduje wykonanie programu aż do osiągnięcia kolejnego wiersza w kodzie, przy czym w odróżnieniu od polecenia *next* po napotkaniu pętli wykonuje ją całą zamiast iteracja po iteracji.

Innym istotnym poleceniem jest polecenie *continue*, które powoduje wznowienie wykonania programu aż do napotkania kolejnego breakpointa lub normalnego zakończenia. Polecenie *finish* z kolei wznawia wykonanie programu do momentu wyjścia z danej funkcji i wyświetla zwróconą wartość, o ile oczywiście taka istnieje [5].

Od wersji 7.0 możliwe jest wykonywanie programu do tyłu (ang. *reverse debugging*). Pozwala to na cofnięcie programu do wcześniejszego punktu, jeśli użytkownik stwierdzi, że zaszło tam coś istotnego dla procesu debugowania. Działanie tej funkcji opiera się na przywracaniu wartości komórek pamięci i rejestrów do wartości sprzed wykonania cofanych instrukcji. Wykorzystywane w tym celu polecenia są odpowiednikami poleceń opisanych w poprzednim akapicie z dodanym przedrostkiem *reverse-*. Dostępne jest w związku z tym polecenie *reverse-step*, które powoduje cofnięcie ostatnio wykonanego wiersza kodu źródłowego, przy czym jeśli było tam zawarte wywołanie funkcji zawierającej informacje dla debuggera, to następuje wejście do tej funkcji i wycofanie wykonania ostatniego jej wiersza. Podobnie jak poprzednio, polecenie *reverse-next* działa analogicznie z tą różnicą, że jeśli wycofywaną instrukcją jest wywołanie funkcji to następuje cofnięcie całej funkcji, a nie tylko jej ostatniej instrukcji. Odpowiednikami tych poleceń dla języka maszynowego są *reverse-stepi* i *reverse-nexti*. Obecne jest również polecenie *reverse-continue*, które rozpoczyna wykonanie programu wstecz i powoduje jego zatrzymanie w momencie napotkania punktów przerwań oraz *reverse-finish*, które powoduje wyjście z funkcji i powrót do miejsca, gdzie była ona wywołana. Istnieje możliwość zmiany działania standardowych poleceń (*step, next* itd.), tak aby działały do tyłu. Służy do tego polecenie *set exec-direction reverse*. Aby powrócić do domyślnego działania należy użyć set exec-direction forward [5].

W analizie stanu programu kluczowe jest poznanie zawartości stosu, na którym podczas każdego wywołania funkcji generowana jest ramka stosu zawierająca dane powiązane z tym wywołaniem. W momencie powrotu z funkcji odpowiadająca jej ramka jest usuwana. Wszystkie ramki składają się na obszar pamięci zwany stosem wywołań. W przypadku architektury x86-64 ramka stosu, jak przedstawiono wcześniej, zawiera parametry wywołania funkcji, adres powrotu, zapisaną poprzednią wartość rejestru RBP (wskaźnik ramki funkcji wywołującej) oraz zmienne lokalne. Jest to w związku z tym bardzo istotna informacja dotycząca funkcjonowania programu. Standardowo polecenia w GDB odnoszą się do ramki aktualnie wykonywanej funkcji, jednak możliwe jest wybranie innej ramki stosu. Można sobie bowiem wyobrazić sytuację, w której

użytkownik chce sprawdzić wartość zmiennej lokalnej jednej funkcji podczas wykonywania innej funkcji. Ramki stosu identyfikowane są w GDB za pomocą numerów, przy czym 0 oznacza ramkę aktualnie wykonywanej funkcji. Kolejne numery identyfikują poprzednie ramki.

Do wyboru aktualnej ramki służy polecenie *select-frame*, w którego parametrze podaje się jej numer lub adres. Podobną rolę spełnia polecenie *frame*, różniące się tym, że wyświetla dodatkowo ogólne informacje o danej ramce. Podanie go bez parametru powoduje wyświetlenie informacji na temat aktualnej ramki. Możliwe jest również użycie polecenia *up* lub *down*, które powodują odpowiednio przejście do poprzedniej lub następnej ramki stosu i wyświetlenie o niej informacji, przy czym w parametrze można podać przesunięcie o odpowiednią liczbę ramek.

Do uzyskania bardziej szczegółowych informacji o ramce może zostać wykorzystane polecenie *info frame*, dzięki któremu można poznać adres danej ramki, adres następnej (wywoływanej) i poprzedniej (wywołującej) ramki, język kodu źródłowego funkcji powiązanej z daną ramką, adres argumentów i zmiennych lokalnych ramki, adres powrotu i rejestry zapisane w ramce.
Możliwe jest też użycie polecenia *info args* do wyświetlenia argumentów wywołania funkcji powiązanej z daną ramką oraz *info locals* do wyświetlenia jej zmiennych lokalnych.

Istotnym poleceniem jest polecenie *backtrace*, które służy do wyświetlenia informacji na temat całego stosu wywołań (bez parametru), określonej liczby ramek z końca stosu (w parametrze liczba dodatnia) lub jego początku (w parametrze liczba ujemna). Informacje o kolejnych ramkach wyświetlane są w kolejnych wierszach w kierunku początku stosu i zawierają numer ramki, nazwę funkcji, licznik programu, nazwę pliku źródłowego, numer wiersza w pliku źródłowym oraz argumenty wywołania funkcji. Pozwala to poznać ścieżkę wywołań funkcji, która doprowadziła do danego miejsca w programie [5].

Do sprawdzania wartości zmiennych wykorzystywane jest polecenie *print*, które w parametrze przyjmuje wyrażenie w języku programowania, w którym napisany jest program. Polecenie to powoduje wyliczenie wartości wyrażenia i wyświetlenie jej w czytelnej formie. Podanie w wyrażeniu zmiennej powoduje wyświetlenie jej wartości, przyczym format dobierany jest odpowiednio do jej typu. Wywołanie polecenia bez parametru powoduje ponowne wyświetlenie wartości poprzedniego wyrażenia.

Jeśli zajdzie potrzeba wyświetlenia zmiennej w innym formacie wyjściowym, można tego dokonać wprowadzając bezpośrednio za nazwą polecenia symbol pożądanego formatu poprzedzony znakiem /. Obsługiwane są następujące formaty wyświetlania:

- x – szesnastkowy,
- d – dziesiętny ze znakiem,
- u – dziesiętny bez znaku,
- o – ósemkowy,
- t – dwójkowy,
- a – adres w postaci szesnastkowej i przesunięcia w stosunku do najbliższego poprzedzającego symbolu,
- c – znakowy wraz z jego numeryczną wartością,
- f – zmiennoprzecinkowy,
- s – łańcuch znaków,
- r – tzw. surowy (raw).

W celu wyświetlenia zawartości całej tablicy dynamicznej należy podać w parametrze wyrażenie w postaci *nazwa tablicy@liczba elementów [3].

Podobną rolę spełnia polecenie *explore*, które umożliwia wyświetlenie informacji o wyrażeniu w sposób interaktywny (od najwyższego poziomu abstrakcji do najniższego). Za jego pomocą można wyświetlać zarówno informacje o wartościach jak i typach danych.

Możliwa jest także bardziej niskopoziomowa analiza zawartości pamięci niezależnie od użytych typów danych. Służy do tego polecenie *x*, które ma składnię podobną do polecenia *print*, jednakże jest kilka rzeczy, na które warto zwrócić uwagę. W przypadku tego polecenia argumentem nie jest bowiem wyrażenie, a adres w pamięci. Oznacza to, że odczytując za jego pomocą wartość zwykłej zmiennej (np. typu *int*) konieczne jest poprzedzenie jej nazwy operatorem adresu, czyli znakiem &. Ponadto użytkownik ma więcej możliwości dotyczących formatu wyświetlania. Obsługiwany jest jeden dodatkowy format oznaczony literką i, który powoduje wyświetlenie zawartości pamięci w postaci instrukcji języka assemblera. Domyślnie używany jest format szesnastkowy.

Dodatkowo użytkownik ma możliwość określenia rozmiaru interesującej go danej (nie dotyczy instrukcji). Dokonuje się tego, podając bezpośrednio za symbolem formatu jeden z następujących symboli rozmiaru:

- b – bajt,
- h – dwa bajty,
- w – cztery bajty,
- g – osiem bajtów.

Bezpośrednio przed symbolem formatu można również określić liczbę powtórzeń – pozwala to na jednoczesne wyświetlenie wielu następujących po sobie danych [3].

Polecenie *display* o składni analogicznej do poprzednich poleceń pozwala na wyświetlanie wartości wyrażenia za każdym razem, gdy program się zatrzymuje. Dzięki temu użytkownik może mieć stałą kontrolę nad wartością jakiejś zmiennej. Polecenie to samo decyduje, czy użyć polecenia *print* czy *x* w zależności od wybranego formatu wyświetlania. Do usuwania pozycji z listy monitorowanych wyrażeń służy polecenie *undisplay* z parametrem definiującym ich indywidualne numery przydzielane przez GDB [3].

W celu wyświetlenia wartości rejestrów możliwe jest użycie powyższych poleceń wraz z parametrem w postaci nazwy rejestru poprzedzonej znakiem *$*, np. *print/x $rax*.

W GDB istnieją cztery standardowe nazwy rejestrów, których można używać niezależnie od architektury. Są to:

- *$pc* (program counter – licznik programu),
- *$sp* (stack pointer – wskaźnik stosu),
- *$fp* (frame pointer – wskaźnik ramki)
- *$ps* (program status – stan programu).

Wartość wszystkich rejestrów za wyjątkiem rejestrów zmiennoprzecinkowych i wektorowych można wyświetlić za pomocą polecenia *info registers*. Wartość pozostałych rejestrów można wyświetlić odpowiednio za pomocą poleceń *info float* i *info vector*. Można też wypisać wartość wszystkich rejestrów bez wyjątku za pomocą polecenia *info all-registers* [5].

W pewnych sytuacjach zachodzi potrzeba wyświetlenia kodu źródłowego. Można tego dokonać za pomocą polecenia *list* wyświetlającego domyślnie 10 wierszy (można to zmienić za pomocą polecenia *set listsize* z odpowiednią wartością określającą liczbę linii). Polecenie to standardowo wyświetla linie otaczające bieżącą instrukcję, a przy następnych wywołaniach kolejne wiersze. Użycie parametru – powoduje wyświetlenie poprzednich linii. Polecenie to umożliwia

także specyfikację pożądanej lokalizacji. Może to być na przykład numer wiersza, przesunięcie w stosunku do ostatnio wyświetlonego wiersza w postaci *+przesunięcie/-przesunięcie*, nazwa funkcji, adres w postaci **adres*, para *nazwa_pliku:numer_linii* i *nazwa_pliku:nazwa_funkcji*. Można też określić zakres linii do wyświetlenia wywołując polecenie w postaci *list pierwsza linia,ostatnia linia*, przy czym nie jest obowiązkowe określenie obu granic [3]. W celu wyświetlenia kodu programu w języku assemblera wraz z adresami poszczególnych instrukcji w pamięci można użyć polecenia *disassemble*. Użycie go z modyfikatorem */m* powoduje wyświetlenie instrukcji języka assemblera wraz z odpowiadającymi im wierszami w kodzie źródłowym, dzięki czemu analiza kodu assemblera staje się prostsza. Modyfikator */r* powoduje z kolei wyświetlenie obok instrukcji assemblera ich reprezentacji szesnastkowej. Domyślnie wypisywany jest kod funkcji odpowiadającej aktualnie wybranej ramce stosu, przy czym instrukcja wskazywana przez licznik programu oznaczana jest za pomocą strzałki.

W przypadku podania w parametrze adresu instrukcji (przy czym można to zrobić np. przez nazwę funkcji) wyświetlany jest kod funkcji, w której dana instrukcja się znajduje. Możliwe jest także ręczne określenie zakresu w postaci *adres_początkowy,adres_końcowy* lub *adres_początkowy,+długość*. Dzięki poleceniu *info line* z opcjonalnym parametrem określającym lokalizację wiersza kodu źródłowego (w postaci podobnej do tej z polecenia *list* za wyjątkiem specyfikacji zakresu) można dowiedzieć się pod jakimi adresami w pamięci znajduje się dany wiersz kodu źródłowego. Pozwala to na szybką analizę kodu w języku maszynowym za pomocą polecenia *x/i* bez konieczności podawania w parametrze adresu [5].

Bardzo istotną funkcjonalnością udostępnianą przez GDB jest możliwość modyfikacji działania programu podczas jego debugowania. Pozwala to w prosty sposób sprawdzić, jaki wpływ na funkcjonowanie programu będzie miała zmiana i czy doprowadzi np. do wyeliminowania znalezionego wcześniej błędu. Możliwa jest przykładowo zmiana wartości zmiennej lub zawartości pamięci. Dokonuje się tego najczęściej za pomocą polecenia set z wyrażeniem w postaci przypisania np. *set nazwa_zmiennej=nowa_wartosc*, jednak aby uniknąć ewentualnych konfliktów nazw zmiennych z ustawieniami GDB zalecane jest stosowanie polecenia set var. W celu zmodyfikowania danych znajdujących się pod konkretnym adresem w pamięci można użyć polecenia w postaci *set {typ_zapisywanej_danej}adres*. Podobne efekty można osiągnąć korzystając z przedstawionego wcześniej polecenia *print* z wyrażeniem przypisania.

Oprócz tego istnieje możliwość wykonania skoku w inne miejsce programu i wznowienie jego wykonania w tym miejscu. Służy do tego polecenie *jump* z parametrem określającym wiersz kodu źródłowego (analogicznie do polecenia *info line*) lub adres instrukcji. Warto przy tym zauważyć, że polecenie to nie zmienia w żaden sposób środowiska wykonania (np. ramki stosu), a jedynie wartość licznika programu. Trzeba to wziąć pod uwagę przy wykonywaniu skoków w dalsze miejsca programu (np. do innej funkcji). Możliwe jest również wcześniejsze, poprawne wyjście z funkcji za pomocą polecenia *return*, które w parametrze przyjmuje wartość zwracaną przez funkcję oraz wywołanie funkcji i wyświetlenie zwróconej wartości za pomocą polecenia *call* z parametrem w postaci wyrażenia wskazującego pożądaną funkcję.
Do programu można również wysłać sygnał z wykorzystaniem polecenia *signal* z parametrem w postaci numeru lub nazwy sygnału [5].

W celu przyspieszenia pracy z debuggerem najczęściej używane polecenia mają swoje skrócone nazwy. Są one następujące:

- at – attach,
- r – run,
- q – quit,
- b – break,
- s – step,
- n – next,
- si – stepi,
- ni – nexti,
- c – continue,
- u – until,
- rc – reverse-continue,
- f – frame,
- info f – info frame,
- bt – backtrace,
- p – print,
- l – list,
- j – jump,
- itd.

Ponadto nie jest konieczne wielokrotne wpisywanie tego samego polecenia w celu jego powtórzenia. Wciśnięcie klawisza Enter bez podania polecenia powoduje ponowne wykonanie polecenia wpisanego jako ostatnie [5].

Omówione powyżej polecenia nie wyczerpują całej ich listy, jednak wydają się być najistotniejsze z punktu widzenia programisty. Więcej informacji na temat poleceń debuggera można uzyskać korzystając w GDB z polecenia *help* wraz z ewentualnym parametrem w postaci nazwy polecenia lub tematu pomocy oraz z dokumentu w formacie GNU info – *info gdb* w powłoce [3].

ŚLEDZENIE ALOKACJI PAMIĘCI ZA POMOCĄ *MTRACE* W BIBLIOTECE GNU C

Implementacja funkcji do zarządzania pamięcią w bibliotece GNU C pozwala na wykrywanie wycieków pamięci oraz zwolnień nie zarezerwowanej pamięci i na uzyskanie informacji na temat ich lokalizacji. Aby skorzystać z tego rozszerzenia GNU, należy włączyć w aplikacji specjalny tryb śledzenia alokacji pamięci. W tym celu należy dołączyć do kodu źródłowego plik nagłówkowy *<mcheck.h>*, a następnie przed rozpoczęciem alokacji pamięci (najlepiej na początku funkcji *main()*) wywołać funkcję *mtrace()*. Użytkownik musi również przed uruchomieniem programu zdefiniować zmienną środowiskową *MALLOC_TRACE* zawierającą ścieżkę do pliku, do którego zapisywane mają być informacje. Można to zrobić np. za pomocą polecenia *export MALLOC_TRACE=leak.log*. Jeśli wartość tej zmiennej jest poprawna, a użytkownik ma prawa zapisu do wskazanej lokalizacji, to wszelkie wywołania funkcji *malloc(), realloc(), memalign()* oraz *free()* będą śledzone i zapisywane we wskazanym pliku [4].

Istnieje możliwość wyłączenia śledzenia w dowolnym miejscu programu poprzez wywołanie funkcji *muntrace()*, ale może to prowadzić do nieoczekiwanych wyników, dlatego zazwyczaj nie jest ona wykorzystywana. Należy pamiętać o tym, że program powinien zostać skompilowany z informacjami dla debuggera (opcja -g kompilatora GCC), bo inaczej nie będzie możliwe uzyskanie informacji na temat lokalizacji poszczególnych wycieków pamięci [12].

Uruchomienie przykładowego programu z wyciekiem pamięci napisanego w języku C (listing 1) i jego zatrzymanie za pomocą kombinacji *CTRL + C* (sygnał *SIGINT*) zakończyło się utworzeniem pliku dziennika zawierającego wiele wpisów w następującej postaci:

@ ./przyklad1:[0x4005ed] + 0xe85460 0x8.

Każdy taki wpis odpowiada konkretnej operacji alokacji lub zwalniania pamięci niezależnie od tego, czy miała ona związek z jakimś wyciekiem pamięci. Można z niego odczytać nazwę pliku wykonywalnego, adres instrukcji, rodzaj operacji (+ dla alokacji, – dla zwalniania) oraz rozmiar bloku.

Z pomocą w wyszukiwaniu wycieków pamięci na podstawie pliku dziennika alokacji przychodzi skrypt *mtrace* napisany w języku Perl dostarczany wraz z biblioteką GNU C. Przyjmuje on w parametrach ścieżkę do pliku wykonywalnego i do pliku dziennika [4].

Wynik jego działania przedstawiono na listingu 12. Został on oczywiście skrócony do pierwszego wykrytego wycieku pamięci, ponieważ wpisów było bardzo dużo (raportowany jest każdy utracony blok pamięci).

Listing 12. Wynik działania skryptu mtrace

```
[p@arch mtrace]$ mtrace przyklad1 leak.log

Memory not freed:
-----------------
           Address  Size     Caller
0x0000000000e85460  0x8  at /home/ph/src/przyklad1.c:10
```

Jak widać, tym razem uzyskano o wiele bardziej przydatne informacje. Dla każdego wycieku pamięci wypisywany jest adres utraconego bloku, jego rozmiar oraz lokalizacja wywołania funkcji alokującej (ścieżka do pliku źródłowego wraz z numerem wiersza).

Narzędzie *mtrace* nie najlepiej sprawdza się w przypadku programów napisanych w języku C++, ponieważ nie jest możliwe uzyskanie informacji na temat miejsca wywołania funkcji alokacji. Wynika to z faktu, iż rejestrowane jest wywołanie funkcji znanych z języka C, a nie operatorów języka C++. Wykonanie przykładowego programu z listingu 3.2 spowodowało utworzenie pliku dziennika z wpisami w postaci:

@ /usr/lib/libstdc++.so.6:(Znwm+0x1d)[0x7fbceccb067d] + 0x1492460 0x4

Widać wyraźnie, że zapisane zostały miejsca wywołania funkcji alokujących w bibliotece standardowej języka C++, a nie w programie. Z tego też powodu skrypt mtrace nie jest w stanie wyświetlić czytelnej lokalizacji w pliku źródłowym programu, a jedynie adres instrukcji w bibliotece standardowej języka C++.

KONTROLA SPÓJNOŚCI STERTY W BIBLIOTECE GNU C

W bibliotece GNU C zaimplementowane zostały również mechanizmy kontroli spójności sterty. Ich działanie opiera się na sprawdzaniu poprawności struktury danych alokatora w momencie wywołania funkcji odpowiedzialnych za zarządzanie pamięcią dynamiczną.
Umożliwia to wykrycie błędów zapisu bezpośrednio za lub przed buforem oraz dwukrotnego zwalniania tej samej pamięci [12].

W wielu dystrybucjach systemu GNU/Linux funkcja ta jest domyślnie włączona. Do sterowania nią służy zmienna środowiskowa *MALLOC_CHECK*. Może ona przyjąć następujące wartości:

- 0 – ignorowanie wykrytych uszkodzeń sterty,
- 1 – wypisywanie informacji diagnostycznych na standardowym wyjściu błędów,
- 2 – zatrzymanie programu w momencie wykrycia błędu,
- 3 – wyświetlenie błędu na standardowym wyjściu błędów i zatrzymanie programu. W tym przypadku nie jest wymagana żadna zmiana w kodzie źródłowym programu lub jego ponowna kompilacja [12].

Na listingu 13 przedstawiono wyniki działania tej funkcji na przykładzie programu z przepełnieniem bufora dynamicznego (listing 7).

Listing 13. Przykład użycia MALLOC CHECK na programie z listingu 7

```
 1 [p@arch src]$ gcc -o przyklad3b przyklad3b.c
 2 [p@arch src]$ export MALLOC_CHECK_=1
 3 [p@arch src]$ ./przyklad3b ABCDEF
 4 Bufor: ABCDEF
 5 Liczba znakow: 6
 6 *** Error in `./przyklad3b': free(): invalid pointer:
   0x0000000002047010 ***
 7 [p@arch src]$ export MALLOC_CHECK_=2
 8 [p@arch src]$ ./przyklad3b ABCDEF
 9 Bufor: ABCDEF
10 Liczba znakow: 6
11 Przerwane (core dumped)
12 [p@arch src]$ export MALLOC_CHECK_=3
13 [p@arch src]$ ./przyklad3b ABCDEF
14 Bufor: ABCDEF
15 Liczba znakow: 6
16 *** Error in `./przyklad3b': free(): invalid pointer:
   0x0000000000d4d010 ***
17 ======= Backtrace: =========
```

```
18 /usr/lib/libc.so.6(+0x7ab06)[0x7f6cef6a8b06]
19 ./przyklad3b[0x4006cc]
20 /usr/lib/libc.so.6(__libc_start_main+0xf5)[0x7f6cef64fa
   15]
21 ./przyklad3b[0x400559]
22 ======= Memory map: ========
23 * pominięto *
24 Przerwane (core dumped)
```

Jak widać, błędy zostały wykryte dopiero pod koniec działania programu, w momencie próby zwolnienia pamięci. Oznacza to, że narzędzie to w żaden sposób nie zapobiega nieprawidłowym operacjom zapisu i ich nie lokalizuje, a jedynie pozwala stwierdzić, że taki fakt miał miejsce. To do użytkownika należy znalezienie odpowiedzialnych za to operacji zapisu. Uruchamiając program pod kontrolą tradycyjnego debuggera takiego jak GDB można jednak zlokalizować konkretne wywołanie funkcji zwalniającej pamięć, która wygenerowała błąd. Pozwala to stwierdzić, której zmiennej dynamicznej dotyczy problem, co może ułatwić znalezienie błędnych operacji zapisu. Można zauważyć, że ustawienie zmiennej MALLOC CHECK na wartość 3 powoduje również wyświetlenie dodatkowych informacji w postaci śladu stosu oraz struktury pamięci (pominięto ze względu na zbyt dużą objętość). Bardziej wnikliwe testy pokazały, że wykrywany jest zarówno zapis za jak i przed buforem.

Podobny efekt można uzyskać wywołując w kodzie programu funkcję *mcheck(0)* przed rozpoczęciem korzystania z dynamicznej alokacji pamięci. Deklaracja tej funkcji znajduje się w pliku nagłówkowym <*mcheck.h*>, który trzeba wcześniej dołączyć do pliku źródłowego za pomocą dyrektywy *#include*. Efektem tego będzie wyświetlenie błędu i zakończenie programu w przypadku wykrycia uszkodzenia danych alokatora. Zachowanie to można zmienić przekazując w parametrze funkcji wskaźnik na funkcję, która ma być wywołana w momencie wystąpienia błędu. Do wywoływanej funkcji przekazywany jest parametr typu *enum mcheck_status*, który identyfikuje rodzaj błędu. Jak wspomniano wcześniej, poprawność danych alokatora dla danego bloku sprawdzana jest podczas wywołania funkcji służących do zarządzania pamięcią dynamiczną. Istnieje jednak możliwość wymuszenia sprawdzenia poprawności tych danych poprzez wywołanie funkcji *enum mcheck_status mprobe(void *wskaźnik)*.

Warto zwrócić uwagę na fakt, że zamiast jawnie wywoływać funkcję *mcheck()* w programie, można również połączyć pliki obiektowe programu z biblioteką *libmcheck.a* poprzez przekazanie do linkera opcji *-lmcheck*, co będzie miało ten sam skutek [12]. Na listingu 14 przedstawiono wynik działania tego samego programu z wykorzystaniem wspomnianej biblioteki.

Listing 14. Przykład użycia biblioteki libmcheck.a na programie z listingu 7

```
 1 [p@arch src]$ gcc -o przyklad3b przyklad3b.c -lmcheck
 2 [p@arch src]$ ./przyklad3b ABCDEF
 3 Bufor: ABCDEF
 4 Liczba znakow: 6
 5 memory clobbered past end of allocated block
 6 Przerwane (core dumped)
 7 [p@arch src]$ ./przyklad3b ABC
 8 Bufor: ABC
 9 Liczba znakow: 3
10 memory clobbered before allocated block
11 Przerwane (core dumped)
```

Można zauważyć, że wykryty został zarówno zapis za buforem jak i przed nim. W tym przypadku wyświetlane komunikaty różnią się od poprzednich i mają bardziej czytelną dla człowieka formę. Oba te rozwiązania działają na tej samej zasadzie i powinny wykrywać te same błędy. Zaletą wykorzystania zmiennej środowiskowej *MALLOC_CHECK* jest to, że nie wymaga ponownej kompilacji ani ponownego linkowania programu.

MECHANIZMY OCHRONY STOSU W GCC

W związku z rosnącą liczbą ataków z wykorzystaniem przepełnień bufora na stosie, do kompilatorów GCC wprowadzone zostały mechanizmy jego ochrony znane jako Stack-Smashing Protector (SSP) lub ProPolice. Jest to rozwiązanie opracowane przez Hiroaki'ego Etoha z japońskiego oddziału firmy IBM. ProPolice powstało jako łatka do GCC w wersji 3.x i zostało do niego wprowadzone wraz z wersją 4.1 Stage 2 (2005).

Zasada działania tej funkcji opiera się na idei znanej wcześniej z rozszerzenia StackGuard. Podczas kompilacji do kodu aplikacji automatycznie dodawany jest specjalny kod ochronny, którego zadaniem jest wykrywanie przepełnień bufora za pomocą tzw. kanarków (ang. *canaries*), zmiana ułożenia zmiennych na stosie (umieszczenie buforów za wskaźnikami) oraz skopiowanie wskaźników będących argumentami funkcji do obszarów poprzedzających bufory lokalne w celu

zabezpieczenia wskaźników [10]. Kanarki to specjalne dane kontrolne umieszczane w ramkach stosu podczas wywołania funkcji, a ich wartość jest sprawdzana podczas wyjścia z funkcji. Ich modyfikacja świadczy o wystąpieniu przepełnienia bufora lub uszkodzenia stosu [8].

W celu skorzystania z SSP należy skompilować program z opcją *-fstack-protector*. Włącza to ochronę dla funkcji wykorzystujących tablice znaków o rozmiarze co najmniej 8 bajtów. Aby zobaczyć, które funkcje nie będą chronione można użyć dodatkowej opcji -Wstack-protector. W celu ochrony wszystkich funkcji należy użyć opcji *-fstack-protector-all*. W części dystrybucji (np. Ubuntu, Hardened Gentoo) ochrona jest włączona domyślnie. W takiej sytuacji można ją wyłączyć za pomocą modyfikatora *-fnostack-protector* [6].

Na listingu 15 przedstawiono wynik uruchomienia przykładowego programu z przepełnieniem w obszarze stosu (listing 9) skompilowanego z mechanizmami ochrony stosu.

Listing 15. Efekt uruchomienia programu z listingu 3.9 skompilowanego z opcją ochrony stosu

```
 1  [p@arch src]$ g++ -fstack-protector-all -o przyklad3c
    przyklad3c.cpp
 2  [p@arch src]$ ./przyklad3c ABCDEFGH
 3  ABCDEFGH
 4  [p@arch src]$ ./przyklad3c ABCDEFGHI
 5  ABCDEFGHI
 6  *** stack smashing detected ***: ./przyklad3c

 7  terminated
 8  ======= Backtrace: =========
 9  /usr/lib/libc.so.6(__fortify_fail+0x37)[0x7f111d8698c7]
10  /usr/lib/libc.so.6(__fortify_fail+0x0)[0x7f111d869890]
11  ./przyklad3c[0x4009f5]
    /usr/lib/libc.so.6(__libc_start_main+0xf5)[0x7f111d78ca
12  15]
13  ./przyklad3c[0x4008b9]
14  ======= Memory map: ========
15  * pominięto *
    Przerwane (core dumped)
```

Można zauważyć, że przepełnienie zostało wykryte dopiero po nadpisaniu 4 bajtów za buforem. Właściwie to miałoby to miejsce również po nadpisaniu 3 bajtów, gdyby ostatni bajt miał wartość inną niż 0. Taką samą wartość miał bowiem pierwszy bajt danych kontrolnych. Niemożność wykrycia mniejszych przepełnień wynika z faktu, iż pamięć na stosie jest wyrównywana.

SSP nie chroni poszczególnych zmiennych lokalnych, tylko rejestry odłożone na stos (adres powrotu i wskaźnik ramki). To przed nimi umieszczane są dane kontrolne [10]. Oznacza to, że przepełnienie z jednej zmiennej lokalnej do drugiej również nie zostałoby wykryte przez SSP. Po wykryciu przepełnienia wyświetlony został ślad stosu i mapa pamięci. Nie została jednak wskazana niepoprawna instrukcja.

Uruchamiając błędny program pod kontrolą debuggera GDB, można dowiedzieć się, której funkcji dotyczy problem (błąd jest wykrywany w momencie wyjścia z funkcji). Zadanie zlokalizowania konkretnej instrukcji należy jednak do programisty.

DEBUGGERY PAMIĘCI

Istnieje wiele dodatkowych narzędzi służących do wykrywania błędów związanych z zarządzaniem pamięcią. W niniejszym rozdziale zaprezentowane zostaną debuggery Valgrind, Electric Fence, DUMA i Dmalloc.

VALGRIND

Valgrind jest programem o zupełnie innej filozofii działania niż GNU Debugger. Nie jest bowiem aplikacją służącą do śledzenia wykonania programu, a modularną platformą do dynamicznej instrumentacji w szczególności do debugowania pamięci i profilowania aplikacji. Umożliwia ona tworzenie narzędzi, które w efektywny sposób dogłębnie analizują działanie programu (np. każdej instrukcji). Valgrind jest wykorzystywany przez rzeszę programistów i w wielu projektach m.in. KDE, Gnome, Mozilla i MySQL. Głównym autorem programu jest Julian Seward, a obecnie zajmuje się nim grupa deweloperów. Valgrind podobnie jak GDB wydawany jest na licencji GPL. Dostępny jest na platformy: GNU/Linux (x86, x86-64, ARM, PPC32, PPC64, S390X, MIPS), Android (ARM, x86) i Darwin – Mac OS X (x86, x86-64). W momencie pisania tej pracy najnowszą wersją jest wersja 3.8.1 wydana 18 września 2012 roku [20].

Na pakiet Valgrind składają się jądro i następujący zestaw podstawowych narzędzi (za: [20]):

- Memcheck – narzędzie do wykrywania błędów związanych z zarządzaniem pamięcią;
- Cachegrind – narzędzie do profilowania wykorzystania pamięci cache procesora i układu przewidywania rozgałęzień;

- Callgrind – narzędzie do profilowania rejestrujące wywołania funkcji w postaci grafu wywołań;
- Helgrind – narzędzie do debugowania programów wielowątkowych;
- DRD – alternatywne narzędzie do debugowania programów wielowątkowych o innej zasadzie działania;
- Massif – narzędzie do profilowania sterty (użycia pamięci);
- DHAT – narzędzie do profilowania sterty dostarczające informacji na temat czasu życia bloków pamięci, ich wykorzystania oraz nieefektywności z tym związanych;
- SGCheck – eksperymentalne narzędzie służące do wykrywania przepełnień buforów globalnych i stosowych;
- BBV – eksperymentalne narzędzie generujące BBV (Basic Block Vectors) do analizy z wykorzystaniem narzędzia SimPoint;
- Lackey – przykładowe narzędzie dostarczające kilku podstawowych informacji na temat programu;
- Nulgrind – najprostsze możliwe narzędzie uruchamiające program bez analizy i instrumentacji wykorzystywane głównie przez deweloperów Valgrinda do debugowania i w celach testowych.

Stworzenie nowych narzędzi z wykorzystaniem platformy Valgrind jest o wiele prostsze, ponieważ nie muszą one być pisane od zera. Valgrind udostępnia bowiem podstawowe funkcje instrumentacji, których samodzielna implementacja jest relatywnie trudnym zadaniem. Wpływa to również pozytywnie na wydajność takich narzędzi, ponieważ wykorzystywana jest stale udoskonalana platforma o dojrzałym kodzie. Narzędzia pisane są w języku C, a ich głównym zadaniem jest przetwarzanie fragmentów kodu przekazywanych im przez jądro Valgrinda [16]. W związku z tym, że praca ta dotyczy debugowania, bardziej szczegółowo zostaną omówione jedynie narzędzia Memcheck, Helgrind, DRD i SGcheck.

Na początku zostanie jednak przedstawiona zasada działania Valgrinda, która opiera się na dynamicznej kompilacji i buforowaniu. Valgrind przy uruchomieniu procesu klienckiego podłącza się pod niego stając się jego częścią i rekompiluje jego kod w locie poprzez deasemblację kodu maszynowego do reprezentacji pośredniej przetwarzanej przez narzędzie i powrotną zamianę do kodu maszynowego. Uzyskany w ten sposób kod zapisywany jest do pamięci cache w celu ewentualnego ponownego uruchomienia i jest nazywany translacją. Pod kontrolą Valgrinda wykonywany jest właśnie ten kod, a nie oryginalny kod programu.

Twórcy Valgrinda musieli zmierzyć się z kilkoma problemami projektowymi wynikającymi z tej filozofii działania: połączenie dwóch programów w jeden proces wymaga bowiem odpowiedniego zarządzania wspólnymi zasobami takimi jak rejestry i pamięć oraz dbania o to, by kontrola nad analizowanym programem nie została utracona w przypadku wywołań systemowych, sygnałów itd.

Sposób działania Valgrinda bardzo utrudnia instrumentację programów o samomodyfikującym się kodzie i często wymaga wykonywania specjalnych zabiegów takich, jak na przykład wyłączenie translacji w określonym zakresie adresów w pamięci [16].

Jak wspomniano wcześniej, narzędzie Memcheck służy do analizy programu pod kątem występowania problemów z zarządzaniem pamięcią. Jego działanie opiera się o trzy rodzaje metadanych wykorzystywanych do kontroli poprawności programu: bity poprawności (V), bity adresowalności (A) i rejestracja alokacji pamięci. Bity poprawności określają, czy odpowiadające im bity rejestrów i pamięci zostały zainicjowane (każdemu bajtowi danych odpowiada 8 bitów V). Są one wykorzystywane do wykrycia dostępu do niezainicjowanych danych. Bity adresowalności określają z kolei, czy program może bezpiecznie odwołać się do danego obszaru pamięci, przy czym na każdy bajt pamięci przypada jeden taki bit. Są one aktualizowane podczas alokacji i zwalniania pamięci, a za ich pomocą wykrywany jest dostęp do niepożądanych obszarów pamięci.

Memcheck przechowuje również informacje na temat każdej alokacji pamięci na stercie (adres bloku i funkcja za pomocą której został alokowany). Dzięki temu możliwe jest wykrycie użycia złej funkcji zwolnienia pamięci (np. free() w przypadku alokacji za pomocą *new*), prób zwolnienia nieprawidłowych bloków pamięci oraz wycieków pamięci (poprzez sprawdzenie przy zamykaniu programu, czy istnieją jakieś nie zwolnione bloki pamięci, na które nie wskazuje żaden wskaźnik – obszar pamięci do wyszukiwania wskaźników określany jest na podstawie bitów A).

Oprócz tego, Memcheck wykorzystuje swoją implementację funkcji alokujących i zwalniających pamięć: każdy blok pamięci na stercie znajduje się pomiędzy nieużywanymi obszarami zabronionymi oznaczonymi jako niedostępne. Pozwala to na wykrycie błędów przepełnień bufora.

Implementacja ta opóźnia także zwolnienie pamięci, co zwiększa prawdopodobieństwo wykrycia prób dostępu do zwolnionych obszarów pamięci. Memcheck wykrywa również próby użycia funkcji *memcpy()* i podobnych na nakładających się obszarach pamięci. Jest to zrealizowane poprzez podmianę tych funkcji na własną implementację [16].

Helgrind i DRD to narzędzia służące do wykrywania błędów w programach wielowątkowych korzystających z wątków POSIX. Narzędzia te wykrywają podobne rodzaje błędów, ale różnią się metodą działania i zdarza się, że niektóre błędy są wykrywane tylko przez jedno z nich. Można dzięki nim wykryć problemy takie jak: nieprawidłowe użycie API wątków POSIX (np. próba odblokowania nieprawidłowego muteksu lub muteksu zablokowanego przez inny wątek, zwolnienie pamięci zawierającej zablokowany mutex), zakleszczenie (wątek oczekuje na zwolnienie zasobów zarezerwowanych przez drugi wątek i vice versa), sytuacje wyścigu (wątki próbują uzyskać dostęp do tych samych danych bez wcześniejszego założenia blokady lub innej synchronizacji) i rywalizacja o blokadę (próba dostępu do tej samej blokady przez wiele wątków). Programowanie wielowątkowe nie jest łatwe, dlatego narzędzia te często bywają niezastąpione, ponieważ niezwykle usprawniają proces tworzenia aplikacji [20].

SGCheck jest pewnego rodzaju dopełnieniem narzędzia Memcheck. W przeciwieństwie do niego kontroluje bowiem granice tablic globalnych i stosowych (lokalnych), co pozwala wykryć przepełnienia bufora mające miejsce w tych obszarach. Nie wykrywa natomiast żadnych innych błędów. Jego działanie bazuje na informacjach debugowania DWARF3 zawartych w pliku wykonywalnym dotyczących lokalizacji tablic globalnych i lokalnych. Wykorzystuje technikę heurystyczną do przewidywania, do której z tablic miała się odwoływać dana instrukcja [20].

Większość dystrybucji systemu GNU/Linux posiada pakiet Valgrind w swoich repozytoriach, więc instalacja sprowadza się najczęściej do wydania odpowiedniego polecenia menedżerowi pakietów. Użycie go jest bardzo proste, ponieważ zadaniem użytkownika jest jedynie uruchomienie programu pod jego kontrolą i analiza otrzymanych wyników. W tym celu należy skompilować program z opcją -g w celu dołączenia do niego informacji debugowania. Pozwala to uzyskać bardziej szczegółowe informacje na temat błędów np. numer linii kodu źródłowego, w której on wystąpił. Podobnie jak w przypadku GDB debugowanie programów zoptymalizowanych przez kompilator jest możliwe, ale zalecane jest debugowanie z wyłączoną optymalizacją (opcja -O0 kompilatora GCC), ponieważ uzyskane w ten sposób rezultaty są łatwiejsze do analizy. W przypadku bardziej agresywnej optymalizacji (opcja -O2) Valgrind może zgłaszać błędy, które w rzeczywistości nie występują [20].

Domyślnie wykorzystywanym przez Valgrind narzędziem jest Memcheck. Aby przetestować program pod jego kontrolą wystarczy zatem uruchomić z wiersza poleceń program valgrind, podając w parametrze ścieżkę do programu i ewentualnie parametry do niego [20]. Poniżej przedstawiono przykład użycia tego narzędzia na przykładzie wycieku pamięci w języku C z listingu 1.
W celu uzyskania bardziej szczegółowych informacji na temat wycieku debugger został uruchomiony z opcją *--leak-check=full*.

Listing 16. Wynik działania programu z listingu 1 pod kontrolą narzędzia Memcheck

```
 1 [p@arch src]$ valgrind --leak-check=full ./przyklad1
 2 ==1390== Memcheck, a memory error detector
 3 ==1390== Using Valgrind-3.8.1 and LibVEX; rerun with -h
   for copyright info
 4 ==1390== Command: ./przyklad1
 5 Srednia z 0 i 10 to 5.000000
 6 Srednia z 1 i 10 to 5.500000
 7 * pominięto *
 8 Srednia z 8588 i 10 to 4299.000000^C
 9 Srednia z 8588 i 10 to 4299.000000
10 ==1390==
11 ==1390== HEAP SUMMARY:
12 ==1390==     in use at exit: 68,712 bytes in 8,589
   blocks
13 ==1390==   total heap usage: 8,589 allocs, 0 frees,
   68,712 bytes allocated
14 ==1390==
15 ==1390== 68,712 bytes in 8,589 blocks are definitely
   lost in loss record 1 of 1
16 ==1390==    at 0x4C2C04B: malloc (in
   /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
17 ==1390==    by 0x4005A7: srednia (przyklad1.c:8)
18 ==1390==    by 0x400618: main (przyklad1.c:25)
19 ==1390==
20 ==1390== LEAK SUMMARY:
21 ==1390==    definitely lost: 68,712 bytes in 8,589
   blocks
22 ==1390==    indirectly lost: 0 bytes in 0 blocks
23 ==1390==      possibly lost: 0 bytes in 0 blocks
24 ==1390==    still reachable: 0 bytes in 0 blocks
25 ==1390==         suppressed: 0 bytes in 0 blocks
26 ==1390== For counts of detected and suppressed errors,
   rerun with: -v
27 ==1390== ERROR SUMMARY: 1 errors from 1 contexts (sup-
   pressed: 2 from 2)
```

Po uruchomieniu wyświetlone zostały informacje na temat narzędzia Valgrind/Memcheck i rozpoczęło się wykonanie programu. W związku z tym, że program działa w nieskończoność, został przerwany za pomocą kombinacji CTRL+C (sygnał SIGINT). Jak widać, uzyskano bardzo wiele informacji na temat użycia pamięci na stercie. Z linii 14 i 15 można dowiedzieć się, że podczas wyjścia z programu było 8589 zarezerwowanych bloków pamięci zajmujących łącznie 68712 bajtów, a w czasie całego działania programu dokonano 8589 alokacji o łącznym rozmiarze 68712 bajtów i 0 zwolnień pamięci.

W kolejnych liniach widoczny jest komunikat o bezpowrotnym utraceniu tych bloków pamięci (czyli o wycieku pamięci) oraz ślad wywołania pozwalający zlokalizować miejsce alokacji powodującej wyciek (jest to informacja o adresach instrukcji w pamięci, nazwach funkcji i plikach źródłowych wraz z numerem linii).

Następnie przedstawione jest podsumowanie na temat wycieków bloków pamięci, które zostały podzielone na następujące kategorie:

- definitywnie utracone – brak jakichkolwiek wskaźników do danego bloku,
- pośrednio utracone – blok został utracony, ponieważ wszystkie bloki na niego wskazujące zostały utracone (np. w przypadku utracenia wskaźnika do korzenia drzewa binarnego),
- prawdopodobnie utracone – istnieją wskaźniki do bloku, ale żaden nie wskazuje na początek bufora,
- wciąż dostępne – bloki nie zostały zwolnione jawnie przed wyjściem z programu, ale dostępne są wskaźniki na ich początek (często jest to uzasadnione i nie jest uznawane za błąd),
- stłumione – wycieki, które zostały celowo zignorowane (np. znane wycieki w bibliotekach systemowych) [20].

Na końcu znajduje się podsumowanie błędów zawierające informacje o ich liczbie i liczbie błędów zignorowanych. Numer 1390 występujący w komunikatach Valgrinda oznacza identyfikator procesu (PID).

Analogiczny efekt ma sprawdzenie drugiego programu z wyciekiem pamięci w języku C++ (listing 2).

Listing 17. Wynik działania programu z listingu 2 pod kontrolą narzędzia Memcheck

```
 1 [p@arch src]$ valgrind --leak-check=full ./przyklad2
 2 * pominięto *
 3 ==2223== Command: ./przyklad2
 4 ==2223==
 5 * pominięto *
 6 51^C
 7 51
 8 ==2223==
 9 ==2223== HEAP SUMMARY:
10 ==2223==     in use at exit: 46,488 bytes in 11,622
   blocks
11 ==2223==   total heap usage: 11,622 allocs, 0 frees,
   46,488 bytes allocated
12 ==2223==
13 ==2223== 46,488 bytes in 11,622 blocks are definitely
   lost in loss record 1 of 1
14 ==2223==    at 0x4C2BA77: operator new(unsigned long)
   (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
   linux.so)
15 ==2223==    by 0x4008CE: losujPodzielna() (przy-
   klad2.cpp:10)
16 ==2223==    by 0x400954: main (przyklad2.cpp:24)
17 ==2223==
18 ==2223== LEAK SUMMARY:
19 ==2223==    definitely lost: 46,488 bytes in 11,622
   blocks
20 ==2223==    indirectly lost: 0 bytes in 0 blocks
21 ==2223==      possibly lost: 0 bytes in 0 blocks
22 ==2223==    still reachable: 0 bytes in 0 blocks
23 ==2223==         suppressed: 0 bytes in 0 blocks
24 ==2223==
25 ==2223== For counts of detected and suppressed errors,
   rerun with: -v
26 ==2223== ERROR SUMMARY: 1 errors from 1 contexts (sup-
   pressed: 2 from 2)
```

Ewidentnie widać, że nastąpił wyciek dla alokacji pamięci operatorem *new* w 10 linii kodu źródłowego (funkcja *losujPodzielna()*).

Kolejnym programem, który został skontrolowany za pomocą narzędzia Memcheck, był program z listingu 7, posiadający błąd przepełnienia bufora znajdującego się na stercie. Zostały wykryte zarówno przepełnienia za buforem oraz nadpisanie bajta danych przed nim. Zgłoszone zostały ponadto próby odczytu danych poza granicami bufora. Poniżej przedstawiono komunikaty wyświetlone w momencie wykrycia tych błędów.

Listing 18. Efekt Komunikat Memchecka przy przepełnieniu bufora o jeden bajt

```
1 ==1114== Invalid write of size 1
2 ==1114==    at 0x4C2CBB2: __GI_strcpy (in
  /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
3 ==1114==    by 0x40067B: main (przyklad3b.c:14)
4 ==1114==  Address 0x51e0046 is 0 bytes after a block of
  size 6 alloc'd
5 ==1114==    at 0x4C2C04B: malloc (in
  /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
6 ==1114==    by 0x400634: main (przyklad3b.c:10)
```

Listing 19. Komunikat Memchecka przy nadpisaniu bajta danych przed buforem

```
1 ==1114== Invalid write of size 1
2 ==1114==    at 0x400684: main (przyklad3b.c:15)
3 ==1114==  Address 0x51e003f is 1 bytes before a block
  of size 6 alloc'd
4 ==1114==    at 0x4C2C04B: malloc (in
  /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
5 ==1114==    by 0x400634: main (przyklad3b.c:10)
```

Listing 20. Efekt Komunikat Memchecka przy odczycie bajta danych za buforem

```
1 ==1114== Invalid read of size 1
2 ==1114==    at 0x4E7E364: vfprintf (in /usr/lib/libc-
  2.17.so)
3 ==1114==    by 0x4E84058: printf (in /usr/lib/libc-
  2.17.so)
4 ==1114==    by 0x40069C: main (przyklad3b.c:17)
5 ==1114==  Address 0x51e0046 is 0 bytes after a block of
  size 6 alloc'd
6 ==1114==    at 0x4C2C04B: malloc (in
  /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
7 ==1114==    by 0x400634: main (przyklad3b.c:10)
```

Komunikaty zawierają informację dotyczącą rozmiaru błędnego zapisu lub odczytu wraz z miejscem jego wystąpienia w postaci śladu stosu. Wyświetlany jest także adres pamięci, którego dotyczy ta operacja wraz z informacją na temat położenia względem przekroczonego bufora i miejsca wystąpienia funkcji alokującej ten bufor.

Jak wspomniano wcześniej, narzędzie Memcheck nie jest w stanie wykrywać przepełnień bufora mających miejsce w obszarze zmiennych statycznych i lokalnych. Do sprawdzenia programów z listingów 5 (przepełnienie bufora globalnego) oraz 9 (przepełnienie bufora lokalnego) należało użyć eksperymentalnego narzędzia SGCheck.
Wymagało to uruchomienia programu *valgrind* z opcją *--tool=exp-sgcheck*. W przypadku programu z przepełnieniem bufora globalnego wykryte zostały jedynie błędy odczytu poza granicami bufora. Wnikliwsze testy pozwoliły stwierdzić, że powodem niewykrycia błędnych operacji *strcpy()* było to, iż łańcuchy znaków kopiowane przez funkcję były wprowadzone jako stałe bezpośrednio w kodzie programu.

Po modyfikacji programu tak by łańcuch znaków był pobierany w parametrze programu, błędny zapis został już wykryty. Nadpisanie bajta danych przed buforem nie zostało natomiast wykryte z powodu ograniczeń narzędzia SGCheck. Działanie tego narzędzia opiera się bowiem na przewidywaniu na podstawie poprzednio wykonanych instrukcji, do której tablicy miała się odwoływać dana instrukcja. Nie jest w związku z tym możliwe wykrycie pojedynczej operacji zapisu poza granicami bufora.

W przypadku programu z przepełnieniem bufora lokalnego (listing 9) wszystkie obecne w nim błędy zostały wykryte nawet w przypadku niewielkiego przekroczenia granic bufora. Komunikaty narzędzia SGCheck miały postać analogiczną do przedstawionej na listingu 21.

Listing 21. Efekt Komunikat Memchecka przy odczycie bajta danych za buforem

```
1 ==2588== Invalid write of size 1
2 ==2588==    at 0x4C2BC9C: __GI_strcpy (in
  /usr/lib/valgrind/vgpreload_exp-sgcheck-amd64-linux.so)
3 ==2588==    by 0x400934: main (przyklad3c.cpp:10)
4 ==2588==  Address 0x7ff0006d6 expected vs actual:
5 ==2588==  Expected: stack array "bufor" of size 6 in
  frame 1 back from here
6 ==2588==  Actual:   unknown
7 ==2588==  Actual:   is 0 after Expected
```

Widać, że komunikaty narzędzia SGCheck są bardzo podobne do tych z narzędzia Memcheck. W tym przypadku mamy do czynienia z zapisem jednego bajta za buforem lokalnym. W komunikatach zawarta jest informacja o oczekiwanym i faktycznym miejscu zapisu. Gdyby przepełnienie to nastąpiło do obszaru innej zmiennej, zostałaby wyświetlona jej nazwa. Jeśli chodzi o przepełnienia bufora na stosie, to w odróżnieniu od narzędzia SSP zaimplementowanego w kompilatorach GCC, narzędzie SGCheck jest w stanie wykryć przepełnienia do innych zmiennych. Ponadto wykrywa błędy w momencie ich wystąpienia, a nie dopiero przy wyjściu z funkcji.

ELECTRIC FENCE

Electric Fence to debugger pamięci przeznaczony dla Linuksa i Uniksa autorstwa Bruce Perensa. Jest to biblioteka zastępująca funkcje biblioteki standardowej C służące do zarządzania pamięcią (*malloc(), realloc(), calloc(), valloc(), free()*) własną implementacją pozwalającą wykrywać błędy związane z dostępem do pamięci na stercie. Wykrywane błędy to: przekroczenia bufora dynamicznego na jego końcu lub początku oraz próby dostępu do zwolnionych bloków pamięci. Wykorzystywane są w tym celu mechanizmy pamięci wirtualnej i sprzętowej ochrony pamięci.

Działanie biblioteki opiera się na umieszczaniu bezpośrednio za lub przed zarezerwowanym buforem niedostępnej dla procesu strony pamięci wirtualnej (bufor umieszczany jest odpowiednio na końcu lub na początku strony pamięci dostępnej dla procesu). Podobnie zwolniona pamięć staje się niedostępna dla procesu poprzez mechanizm stronicowania pamięci. Wszelkie próby dostępu programu do niedostępnych obszarów pamięci, niezależnie od tego, czy jest to odczyt czy zapis, kończą się naruszeniem ochrony pamięci (segmentation fault) i zabiciem procesu. Oznacza to, że program zatrzymuje się dokładnie w miejscu, w którym znajduje się błędna instrukcja. Kolejne błędy mogą zostać wykryte dopiero po naprawieniu poprzednich.

Electric Fence nie wykrywa przepełnień występujących w obszarach zmiennych globalnych i lokalnych oraz wycieków pamięci [4]. W momencie pisania pracy najnowszą dostępną wersją jest wersja 2.2.4.

Wykorzystanie Electric Fence jest bardzo proste. Wystarczy w tym celu dołączyć bibliotekę libefence.a do programu. Nie jest przy tym konieczna powtórna kompilacja, ponieważ jest to wykonywane w procesie konsolidacji (łączenia). Zazwyczaj dokonuje się tego poprzez przekazanie argumentu -lefence do linkera lub poprzez ustawienie zmiennej środowiskowej *LD_PRELOAD*

=libefence.so.0.0 przed uruchomieniem programu (konsolidacja dynamiczna) [3]. Należy przy tym pamiętać, że nie można korzystać jednocześnie z więcej niż jednej biblioteki służącej do debugowania pamięci.

Na listingu 22 przedstawiono efekt działania programu z przepełnieniem bufora na stercie (listing 7) pod kontrolą biblioteki Electric Fence.

Listing 22. Efekt Wynik działania programu z listingu.7 pod kontrolą Electric Fence

```
[p@arch src]$ gcc -O0 -g -o przyklad3b przyklad3b.c -lefence
[p@arch src]$ ./przyklad3b ABCDEFGH

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Naruszenie ochrony pamięci (core dumped)
```

Jak widać, zostało zgłoszone naruszenie ochrony pamięci, co oznacza, że przepełnienie bufora zostało wykryte. W związku z tym, że Electric Fence nie wskazuje dokładnie miejsca tego błędu, a program zatrzymuje się dokładnie w momencie jego wystąpienia, do jego zlokalizowania można wykorzystać debugger GDB. W tym właśnie celu kompilację wykonano z opcją *-g*, ponieważ sama biblioteka tego nie wymaga.

Listing 23. Efekt Debugowanie programu z wykorzystaniem GDB i Electric Fence

```
[p@arch src]$ gdb ./przyklad3b
GNU gdb (GDB) 7.5.1
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /media/sf_src/przyklad3b...done.
(gdb) run ABCDEFGH
Starting program: /media/sf_src/przyklad3b ABCDEFGH
warning: Could not load shared library symbols for

linux-vdso.so.1.
Do you need "set solib-search-path" or "set sysroot"?
```

```
[Thread debugging using libthread_db enabled]
Using host libthread_db library
"/usr/lib/libthread_db.so.1".

Electric Fence 2.2 Copyright (C) 1987-1999 Bruce
Perens <bruce@perens.com>

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff796ccc9 in __strcpy_ssse3 () from
/usr/lib/libc.so.6
(gdb) bt
#0  0x00007ffff796ccc9 in __strcpy_ssse3 () from
/usr/lib/libc.so.6
#1  0x00000000004007dc in main (argc=2,

argv=0x7fffffffe828) at przyklad3b.c:14
(gdb)
```

Procesor tak jak poprzednio zgłosił wyjątek, jednak tym razem uzyskano informację na temat funkcji powodującej przepełnienie, a za pomocą polecenia *bt* wyświetlono ślad stosu. Umożliwiło to stwierdzenie, że przepełnienie występuje w 14 linii pliku przyklad3b.c w miejscu użycia funkcji *strcpy()*. Podobny efekt można uzyskać analizując pod kontrolą GDB plik zrzutu pamięci (ang. *core dump*) tworzony w momencie nieprawidłowego zakończenia programu.

Z debugowaniem za pomocą biblioteki Electric Fence związanych jest kilka problemów. Aby je lepiej zrozumieć, poniżej przedstawiono efekt uruchomienia programu z niewielkim (1-bajtowym) przepełnieniem.

Listing 24. Wynik działania programu z listingu 7 z niewielkim przepełnieniem

```
[p@arch src]$ ./przyklad3b ABCDEF

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce

Perens <bruce@perens.com>
Bufor: ABCDEF
Liczba znakow: 6
```

Okazuje się, że nie zostało wykryte ani niewielkie przepełnienie bufora, ani nadpisanie jednego bajta danych przed nim (zawarte w kodzie programu). Oba te problemy mają inną przyczynę, dlatego najpierw zostanie omówiony pierwszy z nich. Przepełnienie nie zostało wykryte ze względu na sposób w jaki zorganizowana jest pamięć, a konkretnie z powodu wyrównania pamięci.

Większość współczesnych procesorów wykonując operacje na pamięci, dokonuje tego fragmentami (blokami) o rozmiarze słowa procesora (8 bajtów w przypadku procesora 64-bitowego). Z tego też powodu różne wersje funkcji *malloc()* alokują zwykle obszar pamięci wyrównany w pamięci do adresu będącego pewną wielokrotnością słowa. Dzięki temu dostęp do danej nie przekraczającej rozmiaru słowa maszynowego zawsze wymaga tylko jednej operacji dostępu, co zwiększa wydajność całego systemu. W przypadku braku wyrównania dostęp do takich danych może wymagać dwóch operacji dostępu oraz odpowiedniego przetworzenia bloków pamięci przez procesor.

W bibliotece Electric Fence funkcja *malloc()* domyślnie wyrównuje dane do adresów będących wielokrotnością rozmiaru typu *int* (4 bajty dla architektury x86-64) [3]. Oznacza to, że pomimo iż zażądano alokacji bufora o rozmiarze 6 bajtów, w rzeczywistości zarezerwowane zostało 8 bajtów w celu spełnienia wymogów wynikających z wyrównania pamięci.
Przepełnienie bufora zostanie w związku z tym wykryte dopiero w momencie zapisania do niego co najmniej 9 bajtów.

Biblioteka Electric Fence umożliwia zmianę sposobu w jaki wyrównywane są dane. Można tego dokonać poprzez zmienną środowiskową EF ALIGNMENT. Po jej ustawieniu rezerwowane bloki pamięci są wyrównywane zgodnie z jej wartością. Przypisanie jej wartości 1 powoduje wyłączenie wyrównywania, dzięki czemu możliwe staje się wykrywanie nawet 1-bajtowych przepełnień. Zazwyczaj nie powoduje to większych problemów, ponieważ jądro Linuksa umożliwia dostęp do nie wyrównanych danych. Należy się jednak spodziewać znacznego spowolnienia programu [3]. Na listingu 25 przedstawiono wynik działania tego samego programu z wyłączonym wyrównaniem pamięci.

Listing 25. Wynik działania programu z listingu 7 z niewielkim przepełnieniem

```
1  [p@arch src]$ export EF_ALIGNMENT=1
2  [p@arch src]$ ./przyklad3b ABCDEF
3
4    Electric Fence 2.2 Copyright (C) 1987-1999 Bruce

   Perens <bruce@perens.com>
5  Naruszenie ochrony pamięci (core dumped)
```

Tym razem przepełnienie bufora zostało wykryte. Nadpisanie danych znajdujących się bezpośrednio przed buforem nadal nie jest jednak zgłaszane, co potwierdzono poprzez uruchomienie programu z parametrem nie przepełniającym bufora. Wynika to z faktu, iż Electric Fence nie jest w stanie wykrywać jednocześnie niepożądanego dostępu do pamięci za buforem, jak i przed nim.

Jest to spowodowane wykorzystaniem mechanizmu stronicowania pamięci. W przypadku architektury x86-64 i działającego na niej systemu GNU/Linux strona pamięci ma rozmiar 4kB. W celu wykrycia przepełnienia bufora domyślnie jest on umieszczany na końcu strony pamięci, a następna strona pamięci ustawiana jest jako niedostępna dla procesu, dzięki czemu w momencie przepełnienia bufora procesor zgłasza wyjątek. Obszar znajdujący się bezpośrednio przed zarezerwowanym blokiem należy natomiast do tej samej strony pamięci – przy próbie dostępu do niego żaden wyjątek nie jest zgłaszany.

Biblioteka Electric Fence umożliwia zmianę tego zachowania poprzez zmienną środowiskową EF_PROTECT_BELOW. Ustawienie jej na wartość 1 powoduje umieszczanie niedostępnej strony pamięci bezpośrednio przed buforem, a samego bufora na początku strony pamięci zamiast na jej końcu. Dzięki temu możliwe staje się wykrycie niepożądanych prób dostępu do pamięci przed buforem [4]. Gdy opcja ta jest włączona, zmienna EF_ALIGNMENT jest ignorowana. Poniżej przedstawiono efekt ustawienia tej opcji dla przykładowego programu z parametrem nie powodującym przepełnienia bufora (w kodzie programu znajduje się instrukcja nadpisująca jeden bajt przed buforem).

Listing 26. Wynik sprawdzenia programu z ustawieniem EF_PROTECT BELOW=1

```
1  [p@arch src]$ export EF_PROTECT_BELOW=1
2  [p@arch src]$ ./przyklad3b ABC
3
4    Electric Fence 2.2 Copyright (C) 1987-1999 Bruce

   Perens <bruce@perens.com>
5  Naruszenie ochrony pamięci (core dumped)
```

Jak widać, próba zapisu danych przed buforem została wykryta. Po ustawieniu zmiennej środowiskowej EF_PROTECT_BELOW wykrywanie przepełnień bufora mających miejsce za buforem nie jest już możliwe. W związku z tym program powinien być sprawdzany na oba sposoby.

Inną istotną zmienną środowiskową jest EF_PROTECT_FREE. Przypisanie jej wartości 1 powoduje, że obszar pamięci przekazany do zwolnienia funkcji *free()* staje się niedostępny i nie może być więcej alokowany, dzięki czemu łatwiej jest wykryć próby dostępu do zwolnionych bloków pamięci. Domyślnie pamięć ta również staje się niedostępna, ale może być ponownie zarezerwowana w dalszych częściach programu.

Zmienna EF_ALLOW_MALLOC_0 służy z kolei do określenia, czy dozwolone mają być żądania alokacji obszarów o zerowym rozmiarze. Mimo że jest to zgodne ze standardem ANSI, to jest to często wynikiem błędu w programie. Domyślnie taka sytuacja jest zgłaszana jako błąd. Przypisanie tej zmiennej wartości niezerowej oznacza zezwolenie na takie operacje [19].

Przy korzystaniu z biblioteki Electric Fence należy pamiętać o tym, że zwiększa ona znacznie wykorzystanie zasobów. Wynika to z faktu, że każda alokacja wykorzystuje przynajmniej dwie strony pamięci (jedna dostępna dla procesu, a druga niedostępna).

Biorąc pod uwagę, że w rozpatrywanym przypadku każda strona pamięci ma rozmiar 4kB, wykorzystanie pamięci po dołączeniu biblioteki może wzrosnąć wielokrotnie, zwłaszcza gdy program alokuje wiele małych obszarów. W takich sytuacjach może być konieczne zwiększenie obszaru wymiany. Ze względu na to należy też pamiętać, by nie dołączać biblioteki do ostatecznej wersji programu i wykorzystywać ją wyłącznie w trakcie debugowania [19].

DUMA

DUMA (Detect Unintended Memory Access) to biblioteka do debugowania pamięci oparta (fork) na bibliotece Electric Fence. Jego autorami są Hayati Ayguen i Michael Eddington. Dodatkowe funkcje wprowadzone do biblioteki to m.in.:

- wykrywanie nakładających się na siebie obszarów pamięci przy korzystaniu z funkcji takich jak *strcpy(), memcpy()*;
- wykrywanie wycieków pamięci poprzez śledzenie alokacji pamięci (rejestrowana jest też nazwa i numer wiersza funkcji wywołującej) i sprawdzanie przy wyjściu, czy zarezerwowane bloki pamięci zostały zwolnione;
- wykrywanie niezgodności funkcji alokacji i zwalniania pamięci (np. zwolnienie za pomocą *delete()* przy alokacji za pomocą *malloc()*);

- wykrywanie przepełnień bufora w obszarze tej samej strony pamięci poprzez wypełnienie niewykorzystywanego obszaru danymi kontrolnymi, które są następnie sprawdzane (domyślnie podczas zwalniania bloku pamięci);
- wykrywanie prób zwolnienia zwolnionego już bloku pamięci;
- pełne wsparcie dla operatorów *new, new[], delete, delete[]* w C++;
- dodane wsparcie dla systemów z rodziny Microsoft Windows.

Z biblioteki DUMA korzysta się podobnie jak z biblioteki Electric Fence. Należy w tym celu dołączyć do programu bibliotekę *libduma.a*. Zwykle robi się to poprzez uruchomienie linkera z parametrem *-lduma* lub ustawienie zmiennej środowiskowej *LD_PRELOAD=libduma.so.0.0* przed uruchomieniem programu (konsolidacja dynamiczna). Można również uruchomić skrypt *duma.sh* z parametrem w postaci ścieżki do debugowanego programu [7].

W odróżnieniu od Electric Fence, domyślne wyrównanie w bibliotece DUMA jest ustawiane na najniższą obsługiwaną przez procesor wartość. Oznacza to, że w większości przypadków nie ma potrzeby zmieniania wartości wyrównania (w bibliotece DUMA odpowiada za to zmienna środowiskowa DUMA ALIGNMENT). Jej zmiana może być potrzebna, gdy dany program wymaga pewnej konkretnej wartości wyrównania. Nawet jeśli wyrównanie jest większe niż 1 bajt, to małe przepełnienia niewykraczające poza stronę pamięci wykrywane są podczas zwalniania bloków pamięci poprzez sprawdzenie wartości danych kontrolnych umieszczonych za buforem (w tym przypadku wykryty zostanie jedynie błąd zapisu). Ponadto, w celu poprawienia wykrywalności małych przekroczeń granic bufora, w momencie gdy rezerwowany blok pamięci jest mniejszy od wartości wyrównania, biblioteka używa wewnętrznie mniejszego wyrównania równego największej potęgi dwójki nie większej od rozmiaru bloku [7]. Przykładowo: jeśli wyrównanie zostanie ustawione na 16 bajtów, a zarezerwowany zostanie blok pamięci o rozmiarze 7 bajtów, to użyte zostanie wyrównanie do 4 bajtów.

Na listingu 27 przedstawiono wynik uruchomienia programu z listingu 7 z małym przepełnieniem bufora dynamicznego pod kontrolą biblioteki DUMA (bez zmiany domyślnych ustawień).

Listing 27. Wynik sprawdzenia programu z ustawieniem EF_PROTECT BELOW=1

```
1  [p@arch src]$ gcc -O0 -g -o przyklad3b przyklad3b.c

2  -lduma
3  [p@arch src]$ ./przyklad3b ABCDEF
4  DUMA 2.5.15 (shared library, NO_LEAKDETECTION)
   Copyright (C) 2006 Michael Eddington
5
   <meddington@gmail.com>
6  Copyright (C) 2002-2008 Hayati Ayguen
7  <h_ayguen@web.de>, Procitec GmbH
8  Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

   Naruszenie ochrony pamięci (core dumped)
```

Program zakończył się błędem ochrony pamięci. Po uruchomieniu programu pod kontrolą debuggera GDB widać wyraźnie, że błąd ochrony pamięci zostaje zgłoszony w wyniku działania instrukcji *strcpy()* znajdującej się w 14 linii kodu źródłowego.

Oznacza to, że 1-bajtowe przepełnienie bufora zostało wykryte bez problemów. W przypadku Electric Fence do wykrycia tego błędu wymagana była zmiana wartości wyrównania. Tutaj została ona automatycznie ustawiona na najmniejszą obsługiwaną wartość, czyli 1 bajt.

Podobnie jak w przypadku Electric Fence, istnieje możliwość wykrycia niepożądanego dostępu do danych przed buforem. Służy do tego zmienna środowiskowa DUMA_PROTECT_BELOW, której należy w tym celu przypisać wartość 1. Rezerwowane bloki pamięci są wtedy umieszczane na początku strony (domyślnie na końcu), a niedostępna strona pamięci bezpośrednio przed buforem (domyślnie za nim). Stosowanie w tym przypadku detekcji za pomocą mechanizmu stronicowania pamięci nie jest jednak bezwzględnie konieczne.

Jak wspomniano już wcześniej, w odróżnieniu od biblioteki Electric Fence, w całym niewykorzystanym obszarze strony pamięci umieszczane są dane kontrolne, których wartość jest sprawdzana podczas zwalniania bufora. Oznacza to, że nadpisanie danych przed buforem zostanie wykryte nawet wówczas, gdy zmienna środowiskowa DUMA_PROTECT_BELOW nie będzie ustawiona. Należy przy tym pamiętać, że dotyczy to tylko operacji zapisu.

Do wykrycia próby odczytu danych przed buforem zmiana tego ustawienia jest już konieczna [7].

Na listingu 28 przedstawiono działanie tego samego programu z parametrem niepowodującym przepełnienia (występuje tylko nadpisanie bajta danych przed buforem wykonane w kodzie programu).

Listing 28. Wynik działania programu z nadpisaniem danych przed buforem

```
 1 [p@arch src]$ ./przyklad3b ABC
 2 DUMA 2.5.15 (shared library, NO_LEAKDETECTION)
 3 Copyright (C) 2006 Michael Eddington
   <meddington@gmail.com>
 4 Copyright (C) 2002-2008 Hayati Ayguen
   <h_ayguen@web.de>, Procitec GmbH
 5 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
 6 Bufor: ABC
 7 Liczba znakow: 3
 8 DUMA Aborting: ptr=7f87515e1ffa: detected overwrite of
 9 ptrs no mans land below userSpace
10 Błędna instrukcja (core dumped)
```

Widać wyraźnie, że nadpisanie danych przed buforem zostało wykryte. W związku z tym, że dane kontrolne sprawdzane są domyślnie dopiero podczas zwalniania pamięci, błąd wystąpił pod koniec działania programu, a nie w momencie błędnego zapisu. Oznacza to, że nawet wykonanie tego programu pod kontrolą GDB nie da jasnej odpowiedzi, gdzie znajduje się błąd.

Warto w tym miejscu wspomnieć, że biblioteka umożliwia ustawienie okresowego sprawdzania wszystkich danych kontrolnych. Służy do tego zmienna środowiskowa DUMA_CHECK_FREQ, która określa, co ile operacji alokacji lub zwolnienia pamięci mają być sprawdzane dane kontrolne wszystkich stron pamięci. Domyślnie używana jest wartość 0 (sprawdzanie tylko przy zwalnianiu danego bloku). Ustawienie jej na wartość 1 sprawi, że sprawdzanie danych kontrolnych wszystkich bloków będzie miało miejsce przy każdej operacji alokacji lub zwolnienia pamięci. Może to ułatwić lokalizację błędów, ale prowadzi też do znacznego spowolnienia działania programu. W celu dokładnej lokalizacji najlepiej jest użyć zmiennej DUMA_PROTECT_BELOW, co spowoduje, że program zostanie zatrzymany dokładnie w szukanym miejscu [7].

W bibliotece DUMA, podobnie jak w Electric Fence, występuje zmienna sterująca sposobem postępowania ze zwolnioną pamięcią. Jest to zmienna środowiskowa DUMA_PROTECT_FREE. Przyjmuje ona jednak inne wartości i inne jest jej ustawienie domyślne. Domyślnie przyjmuje ona wartość -1, która powoduje, że zwolniona pamięć staje się niedostępna i nie może być więcej przydzielona ani użyta. Pozwala to wychwycić wszystkie próby dostępu do zwolnionej

pamięci, ale powoduje zwiększone użycie zasobów. Przypisanie zmiennej wartości 0 sprawia, że zwalniana pamięć również staje się niedostępna dla procesu, ale może być ponownie alokowana. W przypadku Electric Fence było to ustawienie domyślne. Poprzez przypisanie zmiennej wartości dodatniej można określić maksymalny sumaryczny rozmiar całego chronionego obszaru w kilobajtach. W bibliotece Electric Fence nie istniała taka możliwość. Podobną rolę spełnia zmienna DUMA_MAX_ALLOC, za pomocą której można określić w kilobajtach, jaki maksymalnie obszar pamięci może być wykorzystany przez pamięć zarezerwowaną i chronioną. Po przekroczeniu tej wartości, chroniona pamięć jest zwalniania i może być ponownie wykorzystana [7].

Z kolei zmienne DUMA_MALLOC_0_STRATEGY oraz DUMA_NEW_0_STRATEGY służą do określenia sposobu postępowania z żądaniami alokacji bloków pamięci o zerowym rozmiarze (odpowiednio dla języka C i C++). Pełnią one podobną rolę do zmiennej EF_ALLOW_MALLOC_0 w Electric Fence, ale przyjmują inne wartości.

Pierwszej zmiennej można przypisać: 0 – zakończenie programu z błędem segmentacji, 1 – zwracanie wskaźnika o wartości NULL, 2 – zwracanie zawsze tego samego wskaźnika do jednej z chronionych stron pamięci, 3 – zwracanie wskaźnika na unikalną, chronioną stronę pamięci (domyślnie). W celu wykrywania tego typu alokacji należy ustawić wartość tej zmiennej na 0 (domyślne zachowanie dla Electric Fence). Zmienna dotycząca operatora *new* przyjmuje tylko wartości 2 i 3, przy czym domyślną wartością jest również 3 [7].

Biblioteka DUMA wykrywa również błędy alokacji funkcji *malloc()*, które normalnie prowadziłyby do zwrócenia wskaźnika o wartości NULL. Wystąpienie takiego błędu powoduje zakończenie programu. Można to wyłączyć przypisując zmiennej środowiskowej DUMA_MALLOC_FAILEXIT wartość 0.

Aby możliwe było wykrywanie wycieków pamięci, należy za pomocą dyrektywy *#include* dołączyć do kodu źródłowego programu plik nagłówkowy *duma.h* (dla języka C) lub *dumapp.h* (dla języka C++) oraz wymusić statyczne połączenie biblioteki *libduma.a* z programem [7]. Można tego dokonać za pomocą jednego z poniższych poleceń:

```
gcc -O0 -g -pthread -o przyklad przyklad.c /usr/lib/libduma.a
gcc -O0 -g -pthread -o przyklad przyklad.c -Wl,-Bstatic -lduma –Wl,\
    -Bdynamic
g++ -O0 -g -pthread -o przyklad przyklad.cpp /usr/lib/libduma.a
g++ -O0 -g -pthread -o przyklad przyklad.cpp -Wl,-Bstatic -lduma -Wl,\
    -Bdynamic
```

W ten właśnie sposób pod kontrolą biblioteki zostały uruchomione przykładowe programy z listingów 1 i 2. Programy zmodyfikowano tak, aby kończyły się po wykonaniu 100 pętli. Jest to spowodowane tym, że sprawdzenie, czy zostały zwolnione wszystkie zarezerwowane bloki pamięci, ma miejsce w momencie wywołania funkcji *atexit()* podczas normalnego zakończenia programu. W przypadku zakończenia programu za pomocą sygnału SIGINT (np. za pomocą kombinacji CTRL+C) funkcja ta nie jest wywoływana. Po zakończeniu obu programów dla każdego wykrytego wycieku pamięci został wyświetlony komunikat w postaci analogicznej do poniższej.

DUMA: ptr=0x7faea96b9ffc size=4 type='scalar new' alloced from przyklad2.cpp(11) not freed

Uzyskane informacje to: adres utraconego bloku, jego rozmiar oraz nazwa funkcji jakiej użyto do alokacji i miejsce jej wywołania (nazwa pliku źródłowego wraz z numerem linii). Na końcu wyświetlone zostało również podsumowanie informujące o liczbie wykrytych wycieków pamięci.

DUMA Aborting: DUMA: Reported 100 leaks. There are 0 extra leaks without allocation information

Jak widać, w komunikacie podawana jest także liczba wycieków, na temat których brak jest szczegółowych informacji (np. z powodu nie załączenia pliku nagłówkowego biblioteki do kodu źródłowego lub gdy wyciek występuje w bibliotece systemowej).

Tak jak w przypadku Electric Fence, biblioteka DUMA nie nadaje się do wykrywania błędów przepełnień bufora mających miejsce w obszarze pamięci statycznej i stosu.

Wiążą się z nią również podobne ograniczenia dotyczące wykorzystania zasobów. Konieczność wykorzystania co najmniej dwóch stron pamięci dla każdej alokacji radykalnie zwiększa zapotrzebowanie na pamięć. Z tego też powodu należy pamiętać, by nie dołączać biblioteki do ostatecznej wersji programu.

DMALLOC

Dmalloc jest kolejną biblioteką służącą do debugowania pamięci podmieniającą funkcje takie jak *malloc(), realloc(), calloc(), free()* na swoje implementacje. W odróżnieniu jednak od bibliotek Electric Fence i DUMA, do wykrywania przepełnień bufora nie są wykorzystywane sprzętowe mechanizmy ochrony pamięci. Zamiast tego bezpośrednio za i przed zarezerwowanym buforem umieszczane są specjalne dane kontrolne, których wartość jest weryfikowana.

Oznacza to, że wykrywany jest zapis poza granice bufora, ale odczyt już nie. Jest to spora różnica w porównaniu do poprzednich narzędzi.

Dzięki mechanizmom śledzenia alokacji i zwalniania pamięci, biblioteka ujawnia również wycieki pamięci występujące w programie. Wykrywane są także próby zapisu do zwolnionych już obszarów pamięci. Jest to dokonywane poprzez zapis specjalnych danych do bloku pamięci po jego zwolnieniu w celu ich późniejszej weryfikacji. Biblioteka dostarcza wielu informacji statystycznych dotyczących użycia pamięci, a w przypadku wystąpienia błędu podaje miejsce jego wystąpienia (nazwa pliku i numer wiersza kodu źródłowego).

Biblioteka jest przenośna i działa m.in. na następujących systemach: AIX, BSD/OS, DG/UX, Free/Net/OpenBSD, GNU/Hurd, HPUX, Irix, Linux, MS-DOG, NeXT, OSF, SCO, Solaris, SunOS, Ultrix, Unixware i Microsoft Windows. Jej twórcą jest Gray Watson [21].

Aby skorzystać z biblioteki po jej poprawnej instalacji w systemie, należy dodać odpowiedni alias do pliku konfiguracyjnego powłoki. Użytkownicy najszerzej rozpowszechnionej w systemach GNU/Linux powłoki Bash mogą tego dokonać w pliku konfiguracyjnym *.bashrc* znajdującym się w katalogu domowym. Należy dodać do niego następującą linię:

function dmalloc { eval `command dmalloc -b $`; }

Następnie należy uruchomić ponownie terminal lub wprowadzić to samo polecenie w linii komend.

Do kodu źródłowego sprawdzanych programów należy dodać plik nagłówkowy *dmalloc.h* na końcu listy plików dołączanych za pomocą dyrektywy *#include*. Nie jest to konieczne, jednak pozwala bibliotece wyświetlać informacje na temat lokalizacji problemu (nazwa pliku źródłowego i numer linii). Dzięki kompilacji programu z opcją *–DDMALLOC_FUNC_CHECK* (a więc poprzez zdefiniowanie makra za co odpowiada opcja *-D*) biblioteka może również sprawdzać poprawność wszystkich argumentów wielu standardowych funkcji. Skompilowany program należy połączyć z biblioteką poprzez przekazanie opcji *-ldmalloc* (dla języka C) i *-ldmaloccxx* (dla języka C++) do linkera.
Polecenia kompilacji mogą przyjąć w związku z tym następującą formę:

gcc -DDMALLOC FUNC CHECK -o przyklad przyklad.c -ldmalloc (dla języka C)

g++ -DDMALLOC FUNC CHECK -o przyklad przyklad.cpp -ldmalloccxx (dla języka C++)

Przed uruchomieniem programu trzeba jeszcze włączyć funkcję debugowania. Należy w tym celu uruchomić polecenie *dmalloc -l plik logowania -i 100 tag* debugowania. Opcja *-l* określa plik, w którym zostaną zapisane wyniki działania biblioteki. Opcja *–i* określa, co ile wywołań funkcji zarządzania pamięcią ma być sprawdzana spójność całej sterty (mniejsza liczba pozwala wykryć więcej błędów). Parametr *tag* debugowania określa, jakie funkcje debugowania mają być włączone. Standardowo może przyjąć wartości realtime, low, medium lub high (kolejność od najmniejszego debugowania do największego). Inne opcje można sprawdzić uruchamiając program *dmalloc* z opcją *--usage* [21].

Biblioteka nie jest w stanie wykryć przepełnień buforów znajdujących się na stosie i w obszarze danych statycznych, natomiast uruchomienie pod jej kontrolą programu z przepełnieniem bufora dynamicznego (listing 7) dało rezultaty przedstawione na listingu 29.

Listing 29. Wynik działania programu z listingu 7 pod kontrolą Dmalloc

```
p@ubuntuph:~/src$ gcc -O0 -o przyklad3b przyklad3b.c

-ldmalloc
ph@ubuntuph:~/src$ dmalloc -l log -i 100 low
ph@ubuntuph:~/src$ ./przyklad3b ABCDEF
Bufor: ABCDEF
Liczba znakow: 6
debug-malloc library: dumping program, fatal error
   Error: failed UNDER picket-fence magic-number check
(err 26)
Przerwane (core dumped)
```

Jak widać błąd został wykryty. Domyślnie po wykryciu błędu program jest przerywany. W takiej sytuacji wykrycie kolejnych błędów może być możliwe dopiero po naprawieniu poprzednich.

Aby to zmienić można uruchomić polecenie *dmalloc -m error-abort*, dzięki czemu program nie będzie przerywany w momencie wykrycia błędu [21]. Bardziej szczegółowe informacje dotyczące błędu zostały zapisane w pliku logowania.

Listing 30. Raport Dmalloc po uruchomieniu programu z listingu 7

```
 1  p@ubuntuph:~/src$ cat log
 2  1359521768: 4: Dmalloc version '5.5.2' from

 3  'http://dmalloc.com/'
 4  1359521768: 4: flags = 0x4e48503, logfile 'log'
    1359521768: 4: interval = 100, addr = 0, seen # = 0,
 5  limit = 0
    1359521768: 4: threads enabled, lock-on = 0, lock-init
 6  = 2
 7  1359521768: 4: starting time = 1359521768
 8  1359521768: 4: process pid = 2261
 9  1359521768: 4:   error details: checking pointer admin
    1359521768: 4:   pointer '0x7f5799e54fe8' from
10
    'przyklad3b.c:22' prev access 'przyklad3b.c:11'
11  1359521768: 4:   dump of proper fence-bottom bytes:
12  '\033\253\300\300\033\253\300\300'
    1359521768: 4:   dump of '0x7f5799e54fe8'-8:
    '\033\253\300\300\033\253\300XABCDEF\000\336\312\372'
    1359521768: 4: ERROR: free: failed UNDER picket-fence
    magic-number check (err 26)
```

Z raportu można odczytać podstawowe informacje na temat biblioteki i jej ustawień. Dostępne są również informacje na temat procesu oraz szczegółowe informacje dotyczące przepełnienia bufora, takie jak np. adres bufora i lokalizacja funkcji, która wywołała błąd (nazwa pliku źródłowego i numer linii). Zapisywana jest także oczekiwana wartość danych kontrolnych oraz rzeczywista wartość bufora po przepełnieniu wraz z danymi kontrolnymi.

W ostatniej linii można zobaczyć, że błąd nie został wykryty w momencie nadpisania danych kontrolnych, a dopiero podczas wywołania funkcji *free()*. Oznacza to, że to użytkownik sam musi dokładnie zlokalizować instrukcję, która spowodowała przepełnienie bufora. W tym przypadku wykryty został błąd zapisu przed buforem o czym świadczy słowo UNDER. Po usunięciu z kodu programu błędnej instrukcji zapisującej jeden bajt przed buforem, zgodnie z oczekiwaniami zgłoszone zostało błędne nadpisanie danych za buforem. Wyrównanie pamięci nie jest problemem w przypadku tej biblioteki.

Pod kontrolą biblioteki uruchomiono też przykładowe programy z wyciekami pamięci. Tak jak w przypadku biblioteki DUMA, ograniczono do 100 liczbę iteracji pętli w programach. W obu przypadkach błędy zostały wykryte. W związku z tym, że wyniki były podobne, a raporty o wiele dłuższe niż w przypadku przepełnienia bufora, zamieszczono tylko jeden raport dotyczący programu w C++ w skróconej postaci.

Listing 31. Raport Dmalloc po uruchomieniu programu z listingu 2

```
 1 p@ubuntuph:~/src$ cat log
 2 * pominięto *
 3 1359526299: 100: Dumping Chunk Statistics:
 4 1359526299: 100: basic-block 4096 bytes, alignment 8
   bytes
 5 1359526299: 100: heap address range: 0x7fee6e054000 to
   0x7fee6e05f000, 45056 bytes
 6 1359526299: 100:     user blocks: 1 blocks, 2896 bytes
   (6%)
 7 1359526299: 100:    admin blocks: 10 blocks, 40960
   bytes (91%)
 8 1359526299: 100:    total blocks: 11 blocks, 45056
   bytes
 9 1359526299: 100: heap checked 2
10 1359526299: 100: alloc calls: malloc 0, calloc 0,

11 realloc 0, free 0
   1359526299: 100: alloc calls: recalloc 0, memalign 0,
12 posix_memalign 0, valloc 0
13 1359526299: 100: alloc calls: new 100, delete 0
   1359526299: 100:   current memory in use: 400 bytes
14 (100 pnts)
   1359526299: 100:  total memory allocated: 400 bytes
15 (100 pnts)
   1359526299: 100:  max in use at one time: 400 bytes
16 (100 pnts)
17 1359526299: 100: max alloced with 1 call: 4 bytes
   1359526299: 100: max unused memory space: 1200 bytes
18 (75%)
19 1359526299: 100: top 10 allocations:
   1359526299: 100:  total-size  count in-use-size  count
20 source
   1359526299: 100:           0      0           0      0
21 Total of 0
   1359526299: 100: Dumping Not-Freed Pointers Changed
22 Since Start:
   1359526299: 100:  not freed: '0x7fee6e05e9c8|s1' (4
```

```
23 bytes) from 'unknown'
24 * pominięto *
25 1359526299: 100:  total-size  count  source
26 1359526299: 100:           0      0  Total of 0
   1359526299: 100: ending time = 1359526299, elapsed
   since start = 0:00:00
```

Biblioteka udostępnia bardzo wiele cennych informacji dotyczących wykorzystania pamięci i wycieków pamięci. Można odczytać między innymi rozmiar bloku podstawowego (rozmiar strony pamięci), wartość wyrównania, zakres adresów sterty oraz jej rozmiar, informacje na temat wykorzystania bloków pamięci, liczbę wywołań poszczególnych funkcji zarządzających pamięcią i różne statystyki dotyczące wykorzystania pamięci.

Dla każdego wycieku pamięci wypisywany jest komunikat, który zawiera adres utraconego bloku, jego rozmiar i lokalizację funkcji alokującej. Okazało się, że w przypadku programów w języku C++ nie wszystkie informacje są dostępne. O ile w przypadku programów w języku C można odczytać lokalizację funkcji alokującej (nazwę pliku źródłowego i numer wiersza), tak już w przypadku języka C++ nie można uzyskać takiej informacji. Wynika to z faktu, iż w przypadku języka C++ utrudnione jest przekazywanie do biblioteki takich danych, ponieważ do alokacji i zwalniania pamięci wykorzystywane są zwykle operatory, a nie funkcje [21].

PODSUMOWANIE

Niniejsza praca miała na celu porównanie narzędzi do debugowania programów napisanych w językach C i C++ przeznaczonych dla systemu GNU/Linux. Wymagało to zapoznania się ze specyfiką najczęściej popełnianych błędów programistycznych oraz z wieloma materiałami źródłowymi na temat poszczególnych narzędzi. Na potrzeby pracy napisane zostały przykładowe programy, które następnie były badane z wykorzystaniem tych narzędzi.

Przeprowadzone badania literaturowe i praktyczne pozwoliły stwierdzić, że poszczególne narzędzia różnią się między sobą metodą działania i skutecznością wykrywania różnych błędów. Z jednej strony znalazł się GNU Debugger, który sam w sobie nie wykrywa żadnych błędów, tylko pomaga użytkownikowi je wykryć poprzez śledzenie wykonania programu i dostarczanie informacji na temat jego stanu, a z drugiej wszystkie pozostałe narzędzia, które usprawniają proces debugowania, dzięki zastosowaniu różnych mechanizmów wykrywania błędów. Skuteczność tych narzędzi została przedstawiona w tabeli.

Tabela 1. Porównanie skuteczności narzędzi do debugowania pamięci (opracowanie własne)

	mtrace	**mcheck**	**SSP**	**Valgrind**	**EF**	**DUMA**	**Dmalloc**
wycieki pamięci	+	-	-	+	-	+	+
OOB[1] – globalne	-	-	-	+(rw)	-	-	-
OOB[1] – stertowe	-	+(w)	-	+(rw)	+(rw)	+(rw)	+(rw)
OOB[1] – stosowe	-	-	+[5](w)	+(rw)	-	-	-
UMR[2]	-	-	-	+	-	-	-
UAF[3]	-	-	-	+(rw)	+(rw)	+(rw)	+(rw)
DF[4]	+	+	-	+	+	+	+

r – odczyt, w – zapis
[1] Out-of-bounds – przekroczenie granic bufora
[2] Uninitialized memory read – odczyt niezainicjowanych danych
[3] Use-after-free – próba dostępu do zwolnionej pamięci
[4] Double free – podwójne zwolnienie pamięci
[5] nie dotyczy przepełnień do innych zmiennych, a jedynie do rejestrów zapisanych w ramce stosu

Bezapelacyjnie najbardziej wszechstronnym i najskuteczniejszym debuggerem pamięci okazał się Valgrind. Jako jedyny w tym zestawieniu wykrywa wszystkie uwzględnione rodzaje błędów i radzi sobie z przepełnieniami bufora mającymi miejsce w obszarze danych statycznych.
Płynie z tego naturalny wniosek, że to właśnie on powinien być głównym narzędziem wykorzystywanym do debugowania pamięci.

Innym istotnym spostrzeżeniem jest to, że nie wszystkie przebadane narzędzia wspierają w pełni język C++. Narzędzia mtrace i Dmalloc mają bowiem problemy z odpowiednim wskazaniem lokalizacji wywołań operatorów *new* i *delete*.

Ponadto, istnieją różnice w zakresie wymogu ponownej kompilacji lub konsolidacji programu. Narzędzia Valgrind i mcheck (w wariancie wykorzystującym zmienną środowiskową MALLOC_CHECK_) jako jedyne nie wymagają żadnej z tych czynności.

Kolejnym wnioskiem jest to, iż żadne ze zbadanych narzędzi nie daje stuprocentowej pewności, że błędy w zarządzaniu pamięcią w programie nie występują. Brak błędów zgłoszonych przez debugger oznacza jedynie, że dany zbiór danych wejściowych i dana ścieżka programu tego błędu nie spowodowała.

Nie jest natomiast wykluczone to, że inne dane mogą wywoływać błędy. Należy w związku z tym zawsze przy debugowaniu programów pamiętać o przetestowaniu ich na jak największym zbiorze danych wejściowych.

Niniejsza praca może być przyczynkiem do jeszcze bardziej pogłębionych badań obejmujących również swym zakresem komercyjne narzędzia, narzędzia do statycznej analizy programów oraz wykorzystanie debuggerów w procesie tworzenia aplikacji wielowątkowych w związku z dynamicznym rozwojem procesorów wielordzeniowych oraz obliczeń ogólnego przeznaczenia na układach GPU (GPGPU).

LITERATURA

[1] Erickson J., *Hacking. Sztuka penetracji*, Wydawnictwo Helion, Gliwice 2004

[2] Hawrylak P., *Porównanie narzędzi do debugowania programów w językach C/C++ w środowisku Linux*. Praca inżynierska, Wydział Elektrotechniki i Informatyki, Politechnika Lubelska, Lublin 2013

[3] Johnson M. K., Troan E. W., *Oprogramowanie użytkowe w systemie Linux*, WNT, Warszawa 2000

[4] Mitchell M. et al., *Linux. Programowanie dla zaawansowanych*, Wydawnictwo RM, Warszawa 2002

[5] Stallman R. M. et al., *Debugging with GDB: The GNU Source-Level Debugger*. Tenth Edition, GNU Press, Boston 2011

[6] Stallman R. M. et al., *Using the GNU Compiler Collection*, Free Software Foundation, 2010

[7] Aygün H., *D.U.M.A. – Detect Unintended Memory Access*, http://duma.sourceforge.net (dostęp: grudzień 2012)

[8] Cowan C. et al., *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*, DARPA Information Survivability Conference and Exposition. DISCEX '00. Proceedings, Hilton Head 2000, Tom 2, s. 119–129.

[9] Grötker T. et al., *The Developer's Guide to Debugging*, Springer Science+Business Media B.V., Dordrecht 2008.

[10] http://www.research.ibm.com/trl/projects/security/ssp/ (dostęp: grudzień 2012).

[11] King K. N., *Język C. Nowoczesne programowanie*. Wydanie II, Wydawnictwo Helion, Gliwice 2011.

[12] http://www.gnu.org/software/libc/manual/pdf/libc.pdf (dostęp: grudzień 2012).

[13] Matthew N., Stones R., *Beginning Linux Programming*, 4th Edition, Wiley Publishing, Indianapolis 2008.

[14] Matz M.et al., *System V Application Binary Interface AMD64 Architecture Processor Supplement*. Draft Version 0.99.6, http://www.agguro.be/downloads/manuals/pdfs/linux/abi.pdf (dostęp: grudzień 2012).

[15] Naval History and Heritage Command, *Grace Murray Hopper*, http://www.history.navy.mil/photos/pers-us/uspers-h/g-hoppr.htm (dostęp: grudzień 2012).

[16] Nethercote N., *Dynamic Binary Analysis and Instrumentation*, University of Cambridge, 2004.

[17] Patton R., *Testowanie oprogramowania*, Wydawnictwo Mikom, Warszawa 2002.

[18] Shapiro F. R., *The Yale Book of Quotations*, Yale Univ. Press, New Haven 2006.

[19] Ubuntu Manpage: *efence – Electric Fence Malloc Debugger*, http://manpages.ubuntu.com/manpages/precise/man3/libefence.3.html (dostęp: grudzień 2012).

[20] *Valgrind Documentation*, http://valgrind.org/docs/manual (dostęp: grudzień 2012).

[21] Watson G., *Debug Malloc Library*, http://dmalloc.com/docs/latest/dmalloc.pdf.gz (dostęp: grudzień 2012)..

[22] Wilson P. R. et al., *Dynamic Storage Allocation: A Survey and Critical Review*, Memory Management: International Workshop, s. 1–116, Kinross, 1995.

KAMIL KOŁODZIEJCZYK[1], MAŁGORZATA PLECHAWSKA-WÓJCIK[2]

WYKORZYSTANIE METODYKI PROJEKTOWANIA ZORIENTOWANEGO NA UŻYTKOWNIKA NA PRZYKŁADZIE INTERFEJSU BANKU ON-LINE

WSTĘP

Popularyzacja urządzeń mobilnych i sieci społecznościowych znacząco zwiększyła oczekiwania co do jakości napotykanych interfejsów. Użytkownicy szukają w nich czegoś więcej niż funkcjonalności [15]. Aplikacja musi być prosta w obsłudze, przejrzysta i musi skutecznie angażować użytkownika. Projektanci muszą zatem zrozumieć wagę projektowania doświadczeń (ang. *User Experience, UX*), by tworzony przez nich produkt miał szansę zwrócić uwagę użytkowników, i co za tym idzie, odnieść sukces.

Wzrost sprzedaży smartfonów i tabletów sprawia, że aplikacje stają się podstawowym medium interakcji pomiędzy klientem, a marką. W rezultacie postrzeganie marki staje się ściśle powiązane z jakością oferowanych przez nią interfejsów.

Trend ten został zauważony przez banki internetowe, które w ostatnich latach skupiły swoją uwagę na uwzględnieniu projektowania doświadczeń w serwisach transakcyjnych. Powstałe w ten sposób aplikacje kwestionują zasadność powszechnie stosowanych rozwiązań, stawiając na pierwszym miejscu potrzeby klientów [2-6, 14]. Ich projektanci włączają użytkowników w proces tworzenia interfejsu w oparciu o projektowanie zorientowane na użytkownika, najpopularniejszą metodykę *User Experience*.

Celem artykułu jest zaprezentowanie technik wykorzystywanych w projektowaniu zorientowanym na użytkownika (ang. *User Centered Design, UCD*). W ramach opisanego Case Study zaprojektowano interfejs banku online, stawiając użytkowników docelowych w centrum procesu projektowego.

[1]Politechnika Lubelska, Wydział Elektrotechniki i Informatyki,
student kierunku Informatyka
[2] Politechnika Lubelska, Wydział Elektrotechniki i Informatyki, Instytut Informatyki

PROJEKTOWANIE ZORIENTOWANE NA UŻYTKOWNIKA

User Centered Design jest metodyką umożliwiającą tworzenie użytecznych produktów. Proces ten ma charakter multidyscyplinarny. Od projektanta wymaga się umiejętności obserwacji, znajomości psychologii i zasad ergonomii oraz przede wszystkim – empatii. Jego zadaniem jest skupienie swojej uwagi na użytkowniku, zrozumieniu jego potrzeb i problemów. Dodatkowo musi brać pod uwagę możliwości i ograniczenia wykorzystywanych technologii [16]. Powstały w rezultacie projekt realizuje niezbędną funkcjonalność w sposób zrozumiały dla użytkownika.

UCD dzieli się na pięć głównych faz. Do każdej z nich przydzielone zostały metody pomagające w osiągnięciu jak najlepszych rezultatów:

1. Planowanie (np. analiza kontekstowa)
2. Wymagania (np. diagramy przepływu, analiza istniejących rozwiązań)
3. Design (np. prototypowanie low-fidelity, ewaluacja heurystyczna)
4. Implementacja (np. prototypowanie high-fidelity)
5. Testy (np. ewaluacja heurystyczna, testy użyteczności)

W dalszej części rozdziału omówione zostaną metody wykorzystane w studium przypadku.

ANALIZA KONTEKSTOWA

Analiza kontekstowa projektu polega na sporządzeniu raportu opisującego docelową grupę użytkowników, wraz z zestawem najważniejszych zadań, które będą wykonywać w aplikacji. Z pomocą ankiet oraz wywiadów umożliwia zrozumienie ich oczekiwań oraz wymagań co do projektu.

DIAGRAMY PRZEJŚĆ

Diagramy przejść (ang. *Flow Diagram*) prezentują kroki, jakie muszą zostać podjęte w celu realizacji konkretnego zadania. Prezentują fragmenty interakcji pomiędzy użytkownikiem, a systemem. Dzięki nim możliwe jest zidentyfikowanie decyzji, jakie muszą zostać podjęte w zależności od potrzeb lub preferencji. Pozwalają również zauważyć, z których ścieżek użytkownicy najczęściej będą korzystać [11].

EWALUACJA HEURYSTYCZNA

Ewaluacja heurystyczna to metoda inspekcji użyteczności, która pozwala na ocenę interfejsu w oparciu o zestaw ustalonych heurystyk. Osoby dokonujące analizy oceniają zgodność aplikacji z listą wytycznych w celu wyłonienia przeoczeń i nieścisłości w projekcie.

Najpopularniejszym zestawem heurystyk jest lista utworzona przez Jakoba Nielsena [13]:

- Pokazuj status systemu,
- Zachowaj zgodność pomiędzy systemem a rzeczywistością,
- Daj użytkownikowi pełną kontrolę,
- Trzymaj się standardów i zachowaj spójność,
- Zapobiegaj błędom,
- Pozwalaj wybierać zamiast zmuszać do pamiętania,
- Zapewnij elastyczność i efektywność,
- Dbaj o estetykę i umiar,
- Zapewnij skuteczną obsługę błędów,
- Zadbaj o pomoc i dokumentację.

PROTOTYPOWANIE

Prototypowanie to technika polegająca na tworzeniu schematów interfejsów (makiet), w oparciu o scenariusze użycia oraz diagramy przepływu. Ich zadaniem jest umożliwienie przeprowadzenia ewaluacji planowanych rozwiązań przy minimalnym koszcie wprowadzania modyfikacji.

Prototypowanie pozwala wykryć problemy związane z architekturą informacji, układem strony i doborem funkcjonalności [7].

Makiety o niskim odwzorowaniu detali (*low-fidelity*) wykorzystywane są we wczesnych fazach projektowych, do sprawdzenia ogólnych konceptów.
Schematy o wysokim poziomie odwzorowania (*high-fidelity*) swoim wyglądem są zbliżone do końcowej aplikacji. Wykorzystywane są podczas testów z użytkownikami w celu weryfikacji wyglądu i poziomu użyteczności interfejsu.

ANALIZA EKSPERCKA

Analiza ekspercka jest jednym z najpopularniejszych sposobów testowania aplikacji. Ekspert w trakcie pracy z aplikacją sprawdza i ocenia zdefiniowane wcześniej obszary aplikacji, zwracając uwagę na ich zgodność z wytycznymi dotyczącymi projektowania interfejsów (np. heurystykami Nielsena-Molicha) oraz szuka elementów stanowiących potencjalne miejsca problematyczne [9].

Na potrzeby oceny użyteczności interfejsu definiuje się oraz bada obszary interfejsu aplikacji. W każdym z obszarów można zdefiniować szczegółowe podobszary oraz pytania, na jakie powinien odpowiedzieć ekspert w trakcie pracy z aplikacją.

WĘDRÓWKA POZNAWCZA

Wędrówka poznawcza jest jedną z metod testowania użyteczności i jakości GUI. Umożliwia ona sprawdzenie łatwości uczenia się interfejsu podczas pierwszego kontaktu użytkownika z systemem [9].

Metoda ta opiera się na określeniu kilku zadań, które typowy użytkownik wykona podczas pracy z aplikacją [18]. Zadania te mogą być następnie wykonane przez eksperta. Każde zadanie jest podzielone na części składowe (kroki), które są analizowane zgodnie z następującymi pytaniami:

- Czy użytkownik wie, co zrobić w analizowanym kroku?
- Jeżeli działania wykonywane przez użytkownika są realizowane właściwie, to czy on zdaje sobie z tego sprawę?
- Jeżeli działania wykonywane przez użytkownika są realizowane właściwie, to czy odczuwa on, że zbliża się do celu (poprawnego ukończenia zadania)?

Trudność w każdym etapie ocenia się w skali Likerta, w zakresie od 1 do 5, gdzie 1 oznacza "bardzo łatwy", 5 – "bardzo trudny ".

W przeciwieństwie do innych metod oceny jakości GUI, podstawowym celem realizacji wędrówki poznawczej jest badanie przepływu procesów podejmowanych przez użytkownika, a nie oceny poszczególnych stron lub ekranów interfejsu aplikacji.

Wędrówka poznawcza zyskała popularność ze względu na możliwość wykonywania analizy stosunkowo szybko i przy niskich kosztach. Co ważne, może ona być stosowana również na początku fazy projektu.

Jednak należy zauważyć, że według niektórych badań [8] wędrówka poznawcza często umożliwia identyfikację problemów potencjalnych, które mogą dopiero się pojawić zamiast tych faktycznie istniejących.
W związku z tym metoda ta jest zwykle stosowana w połączeniu z innymi metodami analizy interfejsów.

TESTY UŻYTECZNOŚCI

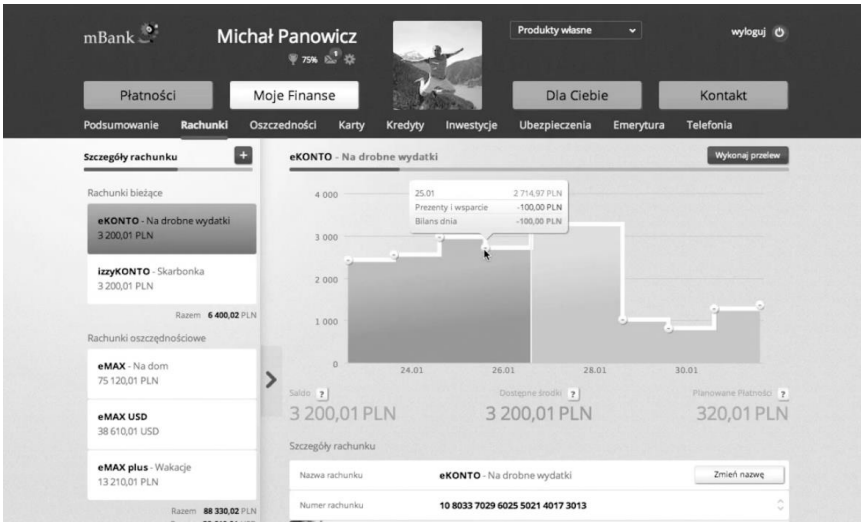
Testy użyteczności stanowią najlepszy sposób na weryfikację skuteczności tworzonego interfejsu. Przeprowadzenie badań pokazuje w jakim stopniu aplikacja jest zrozumiała dla przyszłych użytkowników, wskazując obszary, które mogą sprawiać problemy podczas funkcjonowania strony [7]. W badaniu powinno wziąć udział kilku – kilkunastu użytkowników reprezentujących grupę docelową aplikacji.

Typowy test użyteczności dzieli się na planowanie, wykonanie, analizę i sporządzenie raportu. W celu zapewnienia skuteczności planowanych badań konieczne jest zdefiniowanie najistotniejszych zadań, które zostaną przekazane użytkownikom [1]. W celu osiągnięcia lepszych rezultatów w trakcie testu możliwe jest zastosowanie protokołu głośnego myślenia. Po zakończeniu testów należy sporządzić raport zawierający listę odnalezionych problemów, posortowaną ze względu na przyznane priorytety. Na jego podstawie możliwe jest przeprowadzenie kolejnej iteracji projektowej, poprawiając prototyp i wykorzystując go w kolejnej serii badań użyteczności.

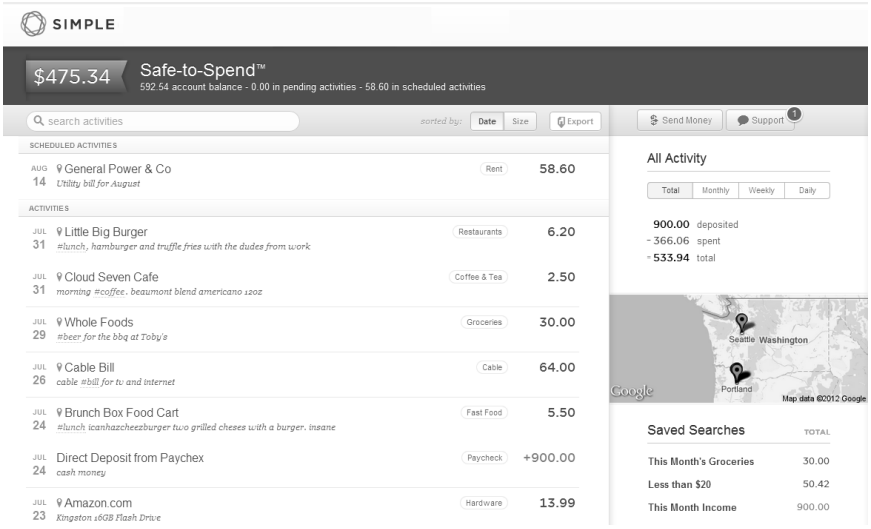
PRZEGLĄD ISTNIEJĄCYCH INTERFEJSÓW BANKOWYCH

Aby możliwe było zrealizowanie projektu przeprowadzona została analiza istniejących rozwiązań, mająca na celu odnalezienie mocnych i słabych stron konkurencyjnych interfejsów. Sprawdzonych zostało 20 aplikacji bankowych. Zostały poddane wędrówce poznawczej, a następnie, wybrane z nich, analizie heurystycznej z wykorzystaniem heurystyk Nielsena. Spośród przeanalizowanych interfejsów dwa wyróżniają się na tle konkurencyjnych rozwiązań (Rys. 1, Rys. 2).

Pierwszym z nich jest aplikacja mBanku. Zaprezentowany na początku czerwca 2013 interfejs miał za zadanie wyznaczyć nowe standardy wśród internetowych aplikacji bankowych. Duża liczba animacji oraz interaktywnych elementów wykazuje na przywiązanie dużej wagi do użyteczności, jednocześnie nawiązując do nowoczesnych serwisów.

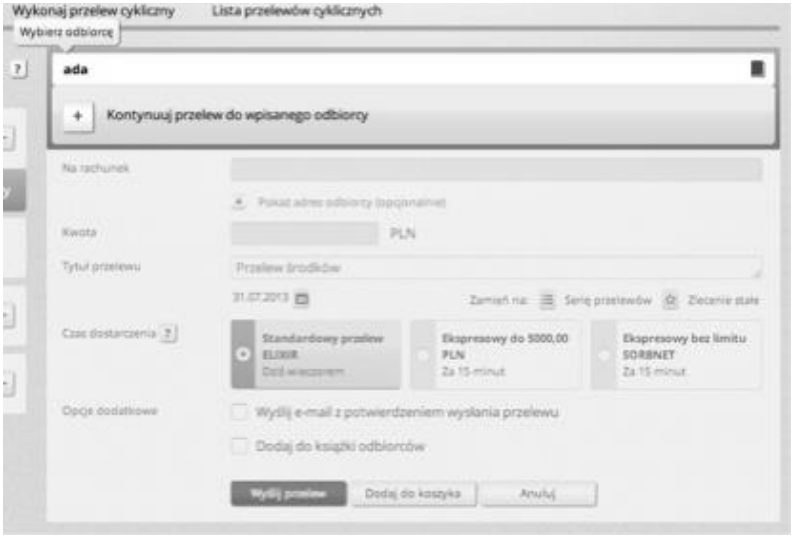


Rys. 1. Interfejs aplikacji mBanku
(źródło: opracowanie własne)



Rys. 2. Interfejs aplikacji Simple
(źródło: opracowanie własne)

System wykonywania przelewów (Rys. 3) składa się z jednego etapu, ograniczając liczbę przeładowań strony. Jedynym aktywnym polem jest nazwa odbiorcy, co nakłania użytkownika do korzystania z książki odbiorców. Pozostała część formularza zostaje zablokowana do czasu wypełnienia pierwszego pola, co może dezorientować początkujących użytkowników. Na dole formularza umieszczony został checkbox umożliwiający dodanie odbiorcy do kontaktów.



Rys. 3. Formularz przelewu w mBanku
(źródło: opracowanie własne)

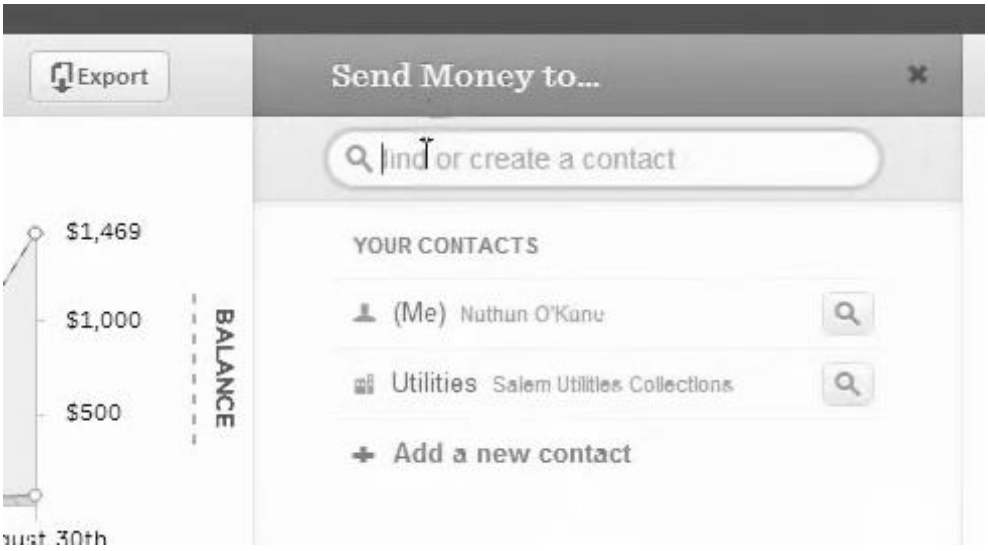
Poza zidentyfikowaniem interesujących rozwiązań utworzona również została lista napotkanych problemów. Są to:

- **Naruszona heurystyka: Pokazuj status systemu**..
  Opis: Podczas wypełniania przelewu architektura informacji sugeruje możliwość kliknięcia na zablokowaną część formularza, w celu przełączenia trybu z książki adresowej na nowy przelew.
- **Naruszona heurystyka: Pokazuj status systemu**.
  Opis: Animacja przejścia pomiędzy podstronami wykonuje się zawsze w tym samym kierunku, niezależnie od aktualnie wybranej przez użytkownika pozycji menu.

- **Naruszona heurystyka: Zapobiegaj błędom.**
  Opis: Pole tytułu przelewu zostaje wypełnione domyślną treścią jeszcze przed zatwierdzeniem formularza. W rezultacie użytkownik może nie wprowadzić swoich danych podczas dokonywania płatności.

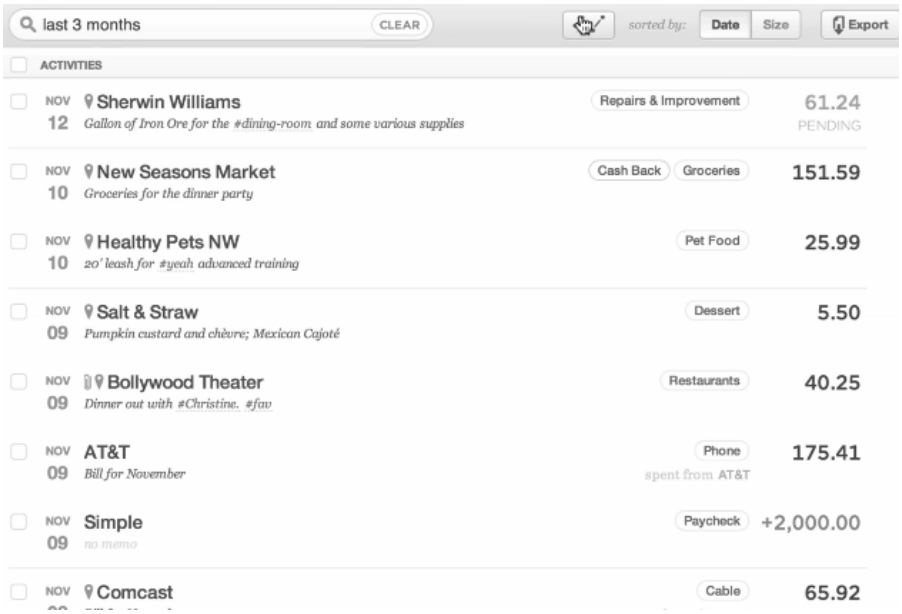
Drugą aplikacją jest amerykański startup, *Simple*. Głównym celem firmy jest dostarczenie klientom konta osobistego wraz z innowacyjnymi usługami. Funkcje takie jak globalny dostęp do przelewów, naturalne grupowanie aktywności, wskaźnik dostępnych środków uwzględniający nadchodzące wydatki wykazują dogłębne zrozumienie potrzeb użytkowników.

Formularz przelewu w tym interfejsie również został zorientowany wokół kontaktów (Rys. 4). Podzielony został na dwa kroki. Przejście pomiędzy nimi odbywa się w sposób dynamiczny, co nie wpływa na efektywność procesu. Pierwsza część umożliwia wybranie kontaktu z listy lub dodanie nowego, podobnie jak w polu nazwa w *mBanku*. Jednak dopiero po pomyślnym wybraniu kontaktu pokazana zostaje dalsza część formularza przelewu. Dzięki temu formularz nie sprawia wrażenia, że pierwszy krok można pominąć.



Rys. 4. Formularz przelewu w Simple
(źródło: opracowanie własne)

W Simple dostępna jest też inteligentna wyszukiwarka, umożliwiająca wprowadzanie zapytań w naturalny sposób (Rys. 5). W prawej części strony znajduje się mapa umożliwiająca filtrowanie wyświetlanych transakcji oraz widok aktywności prezentujący średnie wydatki w poszczególnych okresach czasu. Menu poziome dodatkowo pozwala na wyświetlenie statystyk oraz sortowanie danych. Jego położenie pozostaje niezmienne dla każdej z podstron, jednak funkcjonalność wyszukiwarki i umieszczone obok niej przyciski dostosowują się do aktualnie prezentowanych informacji.



Rys. 5. Wyszukiwarka w Simple
(źródło: opracowanie własne)

Również możliwość swobodnego blokowania i odblokowywania karty płatniczej z poziomu aplikacji jest opcją niezwykle przyjazną użytkownikowi. Ponadto uwagę zwraca nowatorska pomoc techniczna, zrealizowana na wzór komunikatora internetowego, oraz hasła do konta (Passphrase) w formie łatwego do zapamiętania zdania.

WYNIKI

CASE STUDY

Raport kontekstowy, utworzony w wyniku przeprowadzenia analizy kontekstowej, wyróżnił dwa typy użytkowników: posiadacze konta bankowego

oraz osoby zakładające konto bankowe. Na potrzeby projektu uwzględniony zostanie tylko pierwszy z nich. Zebrane informacje przedstawione zostały w Tabeli 1 i Tabeli 2.

Tabela 1. Szczegóły dotyczące wybranego typu użytkowników (opracowanie własne)

	Typ użytkowników	**Posiadacze konta bankowego**
1.2	**Umiejętności i wiedza**	
1.2.1	Doświadczenie w procesach i funkcjach wspieranych przez produkt	Niemal wszyscy zaznajomieni z korzystaniem z konta bankowego
1.2.2	**Doświadczenie w:**	
	korzystaniu z produktów o podobnych funkcjach	Większość dobrze zapoznana z aplikacjami tego typu
	korzystaniu z produktów z takim samym lub podobnym interfejsem	Większość dobrze zapoznana z aplikacjami tego typu
1.3	**Demografia**	
1.3.1	Wiek	Od 14 lat
	Typowy wiek	16 – 65 lat
1.3.2	Płeć	Prawdopodobnie 50% mężczyźni, 50% kobiety
1.4	**Motywacja**	
1.4.1	Podejście do zadań	Bardzo zmotywowani do wykonania zadania
1.4.2	Podejście do produktu	Zróżnicowane
1.5	**Lista zadań**	
	Zidentyfikowane zadania Sprawdzenie historii operacji	Wykonanie przelewu
	Wykonanie przelewu okresowego	Zdefiniowanie przelewu okresowego

Docelowi użytkownicy są dobrze zapoznani z możliwościami internetowych aplikacji bankowych oraz ze sposobem ich obsługi. Przeprowadzone wywiady pozwoliły dodatkowo wyróżnić dwa sposoby korzystania z konta (widoczne w zróżnicowaną częstość występowania przelewu).

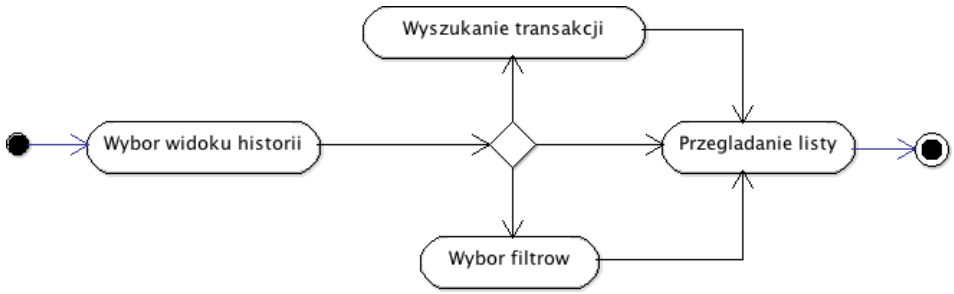
Najdokładniej podsumowuje je podział na podejście: aktywne i pasywne. Pierwsza grupa bardzo intensywnie korzysta z internetowego serwisu transakcyjnego. Wykonują wiele operacji, definiują szablony przelewów oraz ustalają przelewy okresowe. Druga grupa skupia się głównie na sprawdzeniu stanu konta i analizy ostatnich wydatków, po czym wypłacają pieniądze w bankomacie.

Ponadto użytkownicy często przyznawali się do sprawdzania historii operacji po każdym wykonanym przelewie, w celu upewnienia się, czy wszystko wykonało się poprawnie.

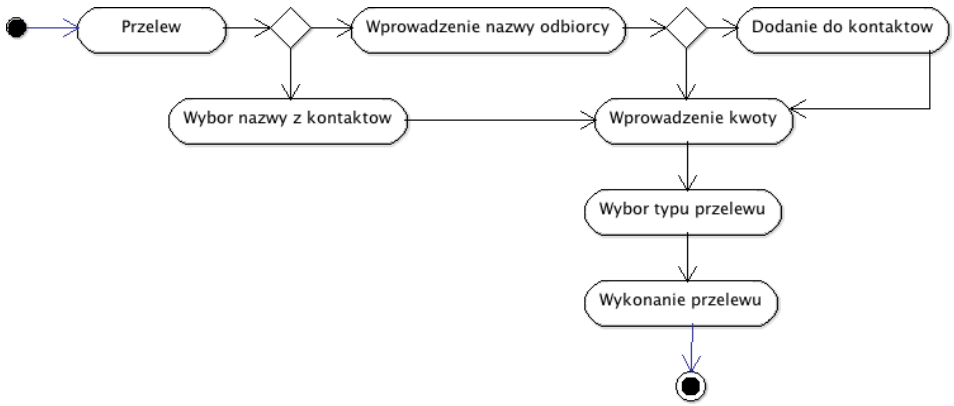
Tabela 2. Wyróżnione zadania (opracowanie własne)

Zadanie		Sprawdzenie historii	Wykonanie przelewu	Analiza wydatków
2	**Charakterystyka**			
2.1	Cel	Sprawdzenie operacji z określonego przedziału czasu	Wykonanie przelewu środków z wybranego konta	Sprawdzenie wydatków w określonym przedziale czasu
2.2	Decyzje	Odnalezienie transakcji poprzez wyszukiwarkę lub przeglądanie listy	Wprowadzenie danych lub wybranie przelewu zdefiniowanego; zdefiniowanie nowego przelewu w trakcie procesu	Brak
2.3	Rezultat	Lista transakcji	Potwierdzenie wykonania przelewu	Wykres prezentujący przychody i wydatki
2.4	Częstość występowania	Często	Zróżnicowana	Zróżnicowana
2.5	Czas trwania	Do 5 minut	Do 5 minut	Do 5 minut
2.6	Zależności	Brak	Posiadanie wystarczających środków na koncie	Brak
2.7	Priorytet (w porównaniu do pozostałych zadań)	Niski	Wysoki	Niski

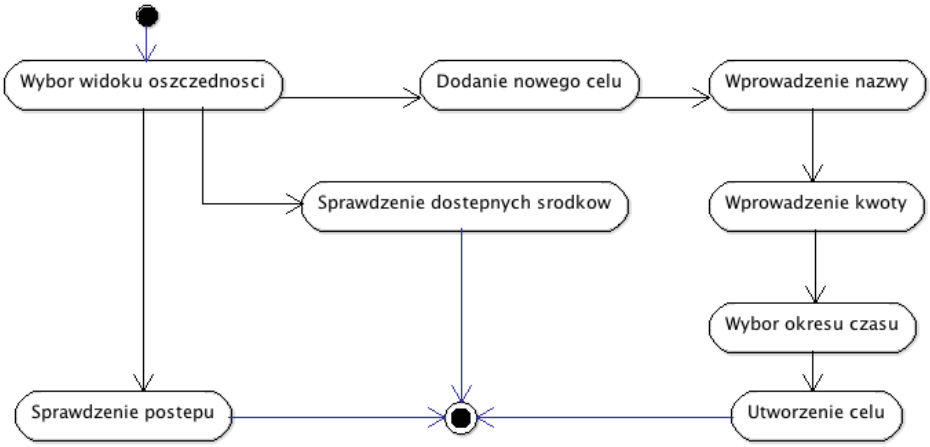
W ramach fazy planowania wymagań, dla każdego z wyróżnionych zadań utworzony został diagram przepływu (Rys. 6–8). Umożliwiły zidentyfikowanie widoków, które muszą zostać zaprojektowane z uwzględnieniem decyzji podejmowanych przez użytkownika podczas realizacji każdego celu.



Rys. 6. Sprawdzenie historii operacji
(źródło: opracowanie własne)



Rys. 7. Wykonanie przelewu
(źródło: opracowanie własne)



Rys. 8. Analiza wydatków
(źródło: opracowanie własne)

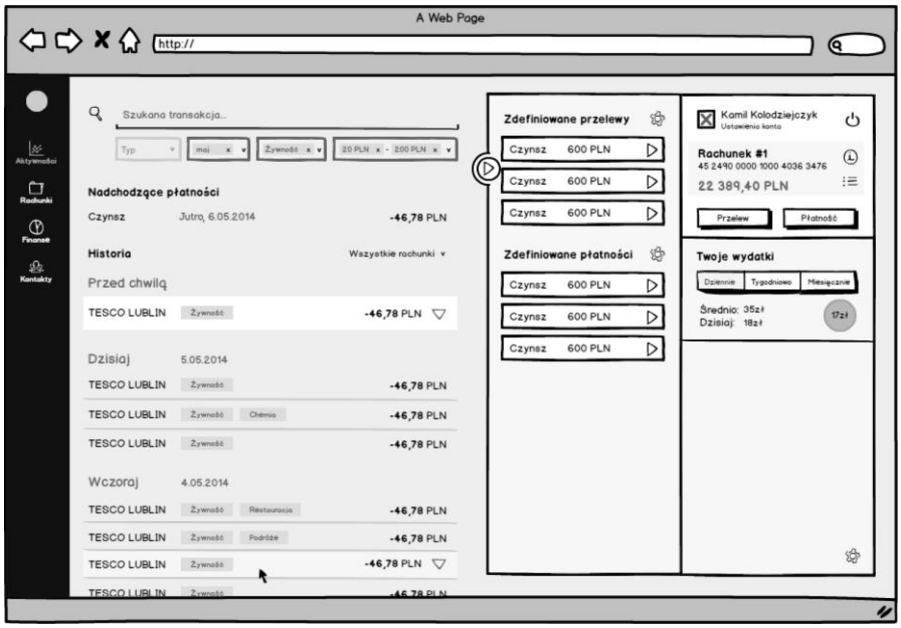
PROJEKT INTERFEJSU

Rezultatem fazy designu było utworzenie dwóch makiet o niskim odwzorowaniu (Rys. 9, Rys. 10). Proponują one odmienne podejścia do realizacji wyznaczonych zadań.

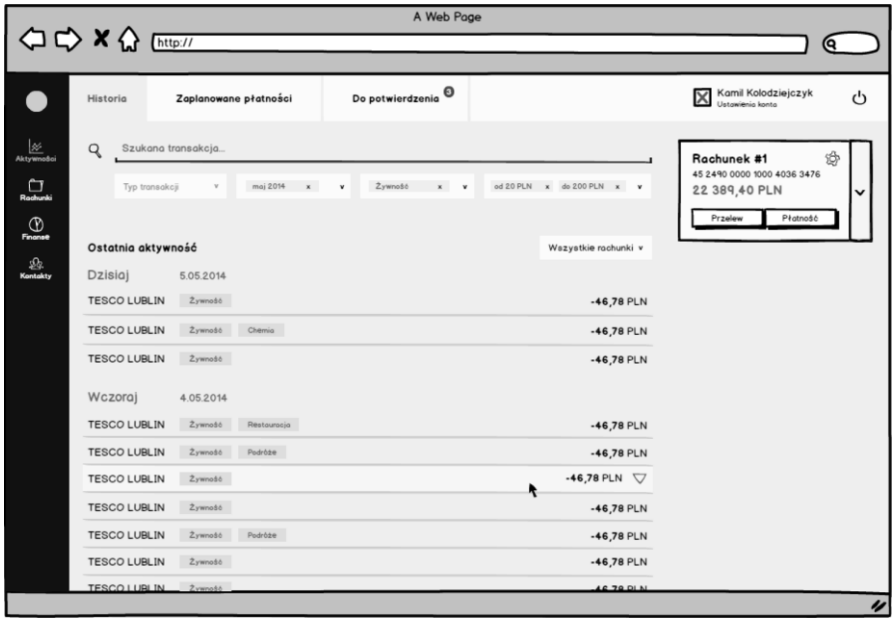
Oba projekty posiadają akceleratory dla aktywnych użytkowników w postaci rozsuwanych *widgetów*. Umożliwiają one szybszy dostęp do bardziej zaawansowanych opcji aplikacji, takich jak szablony przelewu, czy przelewy okresowe.

Widget w pierwszym interfejsie umożliwia spersonalizowanie wyświetlanych przez niego informacji. Jest on znacznie bardziej rozbudowany, co jednak może okazać się przytłaczające dla mniej zaawansowanych użytkowników.

Drugi interfejs charakteryzuje się zmodyfikowaną architekturą informacji w sekcji treści strony. Dodane tu zostało menu poziome zawierające pozycje, które w przypadku pierwszego interfejsu są wyświetlane wewnątrz *widgetu*. Różnica ta pozwoli sprawdzić jak bardzo istotny jest szybki dostęp do ustalonych płatności.



Rys. 9. Pierwsza makieta aplikacji
(źródło: opracowanie własne)



Rys. 10. Druga makieta aplikacji
(źródło: opracowanie własne)

Makiety zostaną poddane analizie eksperckiej z wykorzystaniem listy kontrolnej zamieszczonej w Tabeli 3, w oparciu o skalę ocen zaprezentowaną w Tabeli 4.

Tabela 3. Lista kontrolna przygotowana do badania ankietowego, w ramach analizy eksperckiej (opracowanie własne na podstawie [12])

Interfejs aplikacji
Czy interfejs jest czytelny?
Czy elementy interfejsu są odpowiednio ułożone?
Czy interfejs jest spójny?
Czy jasne jest dlaczego interfejs został zaprojektowany w ten sposób?
Nawigacja
Czy dostęp do wszystkich sekcji jest łatwy i intuicyjny?
Czy poruszanie się po aplikacji jest proste?
Czy menu jest łatwe w użyciu?
Czy łatwo jest znaleźć potrzebne informacje?
Czy nawigacja jest dobrze rozplanowana?
Komunikaty, *feedback*
Czy łatwo jest wykonać wszystkie główne zadania?
Czy wiadome jest, jakie czynności należy wykonać by zrealizować poszczególne zadania?
Treść podstron
Czy etykiety i nagłówki są łatwe do zrozumienia?
Czy treść podstron są zrozumiała?
Czy aplikacja oferuje wszysktkie funkcje, które uważasz za niezbędne?
Czy możliwe jest dostosowanie aplikacji do własnych potrzeb lub upodobań?

Tabela 4. Zastosowana skala ocen (opracowanie własne na podstawie [12])

Ocena	Opis
1	Wystąpiły krytyczne problemy dotyczące użyteczności, uniemożliwiające korzystanie z aplikacji bądź zniechęcające do korzystania z niej
2	Napotkano poważne problemy dotyczące użyteczności mogące uniemożliwić większości użytkowników realizacje zadań
3	Wystąpiły drobne problemy związane z użytecznością, które pojedynczo nie stanowią utrudnienia dla większości użytkowników, jednak ich nagromadzenie może wpłynąć na jakość pracy użytkownika
4	Zidentyfikowano pojedyncze drobne problemy związane z użytecznością mogące obniżyć jakość pracy z aplikacją (np. słaba czytelność tekstu)
5	Nie stwierdzono problemów związanych z użytecznością ani mających wpływ na jakość pracy użytkownika

Kolejnym etapem procesu będzie przeprowadzenie analizy eksperckiej dla zaprojektowanych makiet. Na podstawie otrzymanych danych utworzony zostanie schemat interfejsu, o wysokim odwzorowaniu detali, przeznaczony do bezpośrednich testów z użytkownikami.

PODSUMOWANIE

Przedstawione Case Study prezentuje wpływ jaki może mieć zastosowanie projektowania zorientowanego na użytkownika na proces tworzenia aplikacji. Omówione techniki umożliwiają tworzenie rozwiązań ułatwiających pracę użytkownikom. Pozwalają na dokładniejsze zrozumienie ich nawyków i oczekiwań.

LITERATURA

[1] Allen, J., Chudley, J., *Smashing UX Design: Foundations for Designing Online User Experiences*, John Wiley & Sons Ltd, Chichester, 2012.

[2] Blomkvist, S., *Towards a Model for Bridging Agile Development and User-Centered Design. Human-Centered Software Engineering – Integrating Usability in the Software Development Lifecycle*. Human-Computer Interaction Series, 8, IV, s. 219–244, 2005.

[3] Chamberlain, S., Sharp, H., Maiden, N., *Towards a Framework for Integrating Agile Development and User-Centered Design. Extreme Programming and Agile Processes in Software Engineering*. Lecture Notes in Computer Science, 4044, s.143–153, 2006.

[4] Dayton, D., Barnum, C., *The Impact of Agile on User-centered Design: Two Surveys Tell the Story*. Technical Communication, 56 (3), 2009.

[5] Detweiler, M., *Managing UCD Within Agile Projects*. Interactions 14, 3, s.40–42, 2007.

[6] Dhir, A., Al-kahtani, M, *A Case Study on User Experience (UX) Evaluation of Mobile Augmented Reality Prototypes*. Journal of Universal Computer Science, vol. 19, no. 8, s.1175–1196, 2013.

[7] Kasperski M., *Projektowanie stron WWW: Użyteczność w praktyce*. Helion, Gliwice, 2008.

[8] Koyani S., Bailey R.W., Nall J.R., *Research-Based Web Design & Usability Guidelines*, Computer Psychology, 2004.

[9] Laskowski M.: *Czynniki zwiększające jakość użytkową interfejsów aplikacji internetowych*, Logistyka 6/2011, Instytut Logistyki i Magazyn. 2011, s. 2191–2199.

[10] Marin, B., *Universal Methods of Design: 100 Ways to Research Complex Problems*, Develop Innovative Ideas, and Design Effective Solutions, Rockport Publishers, Beverly, 2012.

[11] Mathis, L, *Designed for Use*, The Pragmatic Bookshelft, Dallas, 2011.

[12] Milosz M., Plechawska-Wojcik M., Borys M., Laskowski M., *Quality Improvement of ERP System GUI Using Expert Method: a Case Study*, 6th International Conference on Human System Interaction, 2013.

[13] Nielsen, J., *Heuristic evaluation*. [w:] Nielsen, J., and Mack, R.L. (Eds.), Usability Inspection Methods, 1994.

[14] Prenzel A., Ringwelski G., *Design of Human-Computer Interfaces in Scheduling Applications*. Proceedings of ICEIS Conference, s. 219–228, 2012.

[15] Resmini, A., Rosati, L., *Pervasive Information Architecture: Designing Cross-Channel User Experiences*. Morgan Kaufmann Publishers, 2011.

[16] Rubin, J., Chisnell, D., *Handbook of Usability Testing, Second Edition: How to Plan, Design, and Conduct Effective Tests*, Wiley Publ. Inc., Indianapolis, 2008

[17] Thomas, C., Bevan, N., *Usability Context Analysis: a Practical Guide*, National Physical Laboratory, Teddington, 1996.

[18] Wharton C. et al., *The cognitive walkthrough method: A practitioner's guide*. [w:] Usability inspection methods. New York, NY: John Wiley & Sons Inc., 1994.

MACIEJ MIŁOSZ[1], MATEUSZ MICHALCZYK[1], ALEKSANDER WOJDYGA[2]

CLOUD COMPUTING NA PRZYKŁADZIE SALESFORCE

WSTĘP

Termin cloudcomputing pochodzi od angielskich słów *cloud* – chmura i *compute* – obliczać. Odnosi się on do dostępu do infrastruktury, usług, narzędzi poprzez sieć, przeważnie Internet. Zakłada on wykorzystanie platformy sprzętowej i oprogramowania jako usługi, a nie jako produktu.

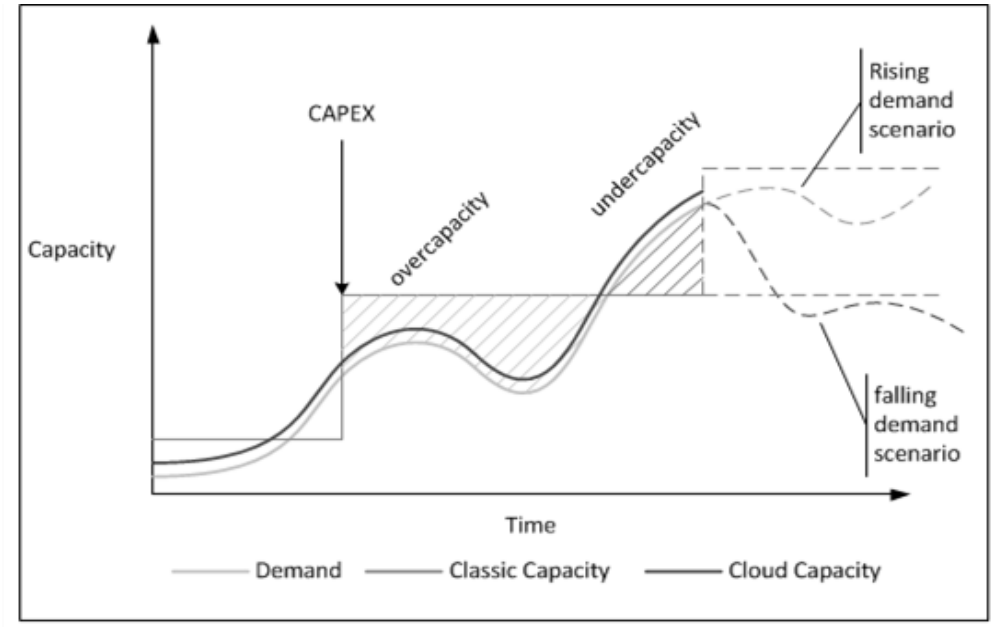
Główną zasadą działania chmury obliczeniowej jest przeniesienie całej infrastruktury IT, na którą się składają dane, moc obliczeniowa i oprogramowanie na serwer i stałe udostępnienie tych zasobów użytkownikom poprzez komputery klienckie. Dzięki temu można połączyć się z serwerem poprzez sieć i korzystać z pełnych zasobów chmury.

Charakterystyczne cechy dla chmur obliczeniowych to:

- Sprzęt jest własnością dostawcy usługi, a nie usługobiorcy.
- Użytkownik nie musi posiadać wiedzy o lokalizacji danych lub procesów, który sprzęt a danym momencie go obsługuje. Istotne jest to z punktu widzenia prawa, ale w tym wypadku odpowiada za to dostawca usługi.
- Dane, aplikacje i procesy często są przenoszone między różnymi fizycznymi jednostkami, w celu zapewnienia maksymalnej wydajności sprzętowej.
- Użytkownicy mają możliwość korzystania z zakupionej usługi poprzez sieć (zwykle Internet), gdziekolwiek i kiedykolwiek poprzez komputery, laptopy czy urządzenia mobile.
- Użytkownicy płacą za to, z czego korzystają, w szczególności moc obliczeniową, pojemność bazy danych czy aplikacje i mają możliwość w łatwy sposób rozszerzyć ilość urządzeń, z których korzystają.

[1] Politechnika Lubelska,Wydział Elektrotechniki i Informatyki,
student kierunku Informatyka
Transition Technologies S. A.
[2] Transition Technologies S. A.

Chmury obliczeniowe mogą być wykorzystywane do przechowywania danych lub korzystania z oprogramowania. Jedynym z największych plusów chmur obliczeniowych jest stały i przewidywany koszt użytkowania. Kolejnym plusem, który wpływa na koszt użytkowania jest elastyczność.



Rys. 1. Porównanie popytu na usługę z przepustowością serwera i chmury (źródło: [9])

Na Rysunku 1 widać jedną z głównych zalet chmur obliczeniowych. W przypadku zmiennego zapotrzebowania, elastyczność chmury umożliwia dostosowanie mocy obliczeniowej i pamięci do aktualnych potrzeb. Możemy wyróżnić trzy główne modele przetwarzania w chmurze: IaaS, PaaS, SaaS.

INFRASTRUCTURE AS A SERVICE (IAAS)

Model Infrastructure as a Service (ang. „infrastruktura jako usługa") polega na dostarczeniu infrastruktury informatycznej użytkownikowi. Zawiera ona sprzęt, oprogramowanie oraz serwis. Klient wykupuje na przykład określoną ilość przestrzeni dyskowej, mocy obliczeniowej.

PLATFORM AS A SERVICE (PAAS)

Model Platform as a Service (ang. „platforma jako usługa") polega na dostarczeniu mocy obliczeniowej, przestrzeni dyskowej i środowiska deweloperskiego. W tym modelu użytkownik sam tworzy oprogramowanie, które używa zasoby dostarczone przez usługodawcę.

SOFTWARE AS A SERVICE (SAAS)

Model Software as a Service (ang. „oprogramowanie jako usługa") polega na zapewnieniu klientowi konkretnych funkcjonalności oprogramowania. Wszystkie aplikacje znajdują się na serwerach usługodawcy, korzystają z ich mocy obliczeniowej i pamięci dyskowej. Klient uzyskuje dostęp do oprogramowania na żądanie.

MULTITENANCY

Termin multitenancy (ang. „wielodostępność") odnosi się do fundamentalnej technologii, z której korzystają chmury obliczeniowe. Polega ona na bezpiecznym i efektywnym kosztowo współdzieleniu zasobów.

Wielodostępność można porównać do mieszkańców bloku, w którym klatka schodowa, drzwi podłączenia mediów są wspólne, ale każdy z nich ma własne mieszkania dające im prywatność. Dzięki współdzieleniu zasobów koszt utrzymania chmury jest mniejszy, a moc obliczeniowa jest w pełni wykorzystywana w każdym momencie działania.

SALESFORCE.COM

Salesforce.com jest firmą, która została założona w 1999 roku przez byłego pracownika firmy Oracle Marca Benioffa, Parkera Harrisa, Dave'aMoellenhoffa i Franka Domingueza. Od początku swojego istnienia firma ta specjalizuje się w modelu chmury SaaS. Według danych z roku 2013 zatrudniała ona ponad 12 tysięcy pracowników.

TECHNOLOGIE SALESFORCE

Niniejsza sekcja opisuje dostępne komponenty do budowy aplikacji na platformie Salesforce. Opisuje dostępne środki do budowy logiki biznesowej i dostosowania interfejsu użytkownika.

Każda aplikacja wdrożona na platformie Salesforce zbudowana zgodnie z założeniami wzorca architektonicznego Model-View-Controller ([4], [5]). Do warstwy modelu należą obiekty standardowe i niestandardowe, do warstwy widoku należą karty obiektów oraz strony Visualforce. Warstwa kontrolera może składać się z reguł walidacji danych, zdefiniowanych przepływów czy procesów zatwierdzania.

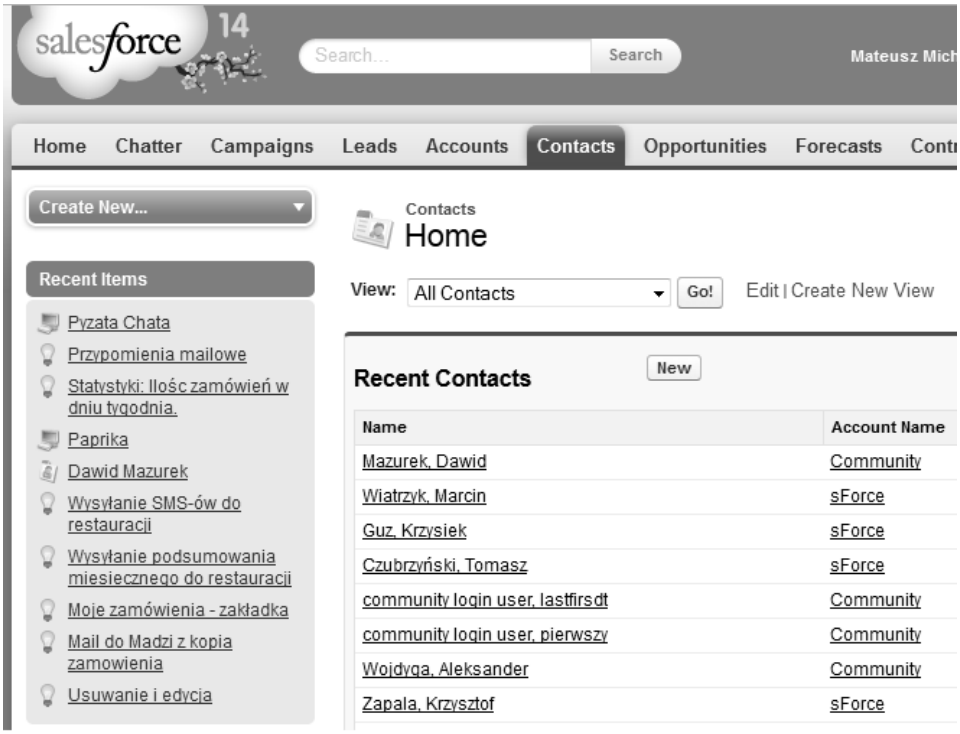
OBIEKTY STANDARDOWE I NIESTANDARDOWE

Za pomocą standardowych obiektów Salesforce zostały zbudowane dwie aplikacje: Sales Cloud i Service Cloud. Zgodnie z pomysłem producenta mieszczą się one w ramach ogólnej usługi Customer Relationship Management, bo taki właśnie segment rynku Salesforce Inc. w tej chwili zajmuje [6].

Za pomocą Sales Cloud realizowana jest obsługa sprzedaży od pozyskiwania klientów przez podtrzymywanie relacji i kampanie marketingowe aż po zarządzanie cenami i stawkami. Obsługa zgłoszeń serwisowych od klientów jest realizowana przez Service Cloud. Klient Salesforce może uruchomić dowolną z tych aplikacji lub jej część w postaci wybranych obiektów. Wszystkie obiekty można wybrać, aby utworzyć własną aplikację odpowiadającą na dane potrzeby biznesowe. Na Rys. 2 przedstawiono kartę obiektu Contacts osadzonego w standardowej aplikacji Sales Cloud.

Obiekty powiązane są ze sobą relacjami jeden do wielu oraz wiele do wielu na podobieństwo relacyjnego modelu danych. Możliwa jest również relacja hierarchiczna łączaca rekordy tego samego obiektu.

Klient ma możliwość tworzenia własnych obiektów modelujących dane przetwarzane w dotyczącym go procesie biznesowym. Składowe tych obiektów (pola) mogą być zawierać proste informacje typu liczbowego, logicznego, napisowego, ale także dane o ustalonym formacie jaka adres email czy URL.
W dwóch ostatnich przypadkach interfejs użytkownika oraz związana z nim logika będzie wymuszać dostarczenie danych w poprawnym formacie przed zapisaniem ich w bazie danych. Jest to jedna z wielu standardowych funkcjonalności platformy działająca bez żadnej konfiguracji czy programowania.



Rys. 2. Sales Cloud
(źródło: opracowanie własne)

STRONY VISUALFORCE

Za pomocą stron Visualforce programista może w swobodny sposób dostosować interfejs użytkownika. Do tworzenia tych stron używany jest język szablonów zawierający znaczniki podobne do języka HTML czy JSP. Z każdą taką stroną skojarzony jest standardowy kontroler umożliwijacy w prosty sposób wykonywanie bazodanowych operacji zapisu czy wyszukiwania..

Dostępne jest szerokie spektrum szczegółowości komponentu Visualforce, od sekcji strony po pojedyncze pole obiektu. W treści strony mogą znajdować się znaczniki Visualforce jak <apex:dataTable> czy <apex:enhancedList> a także dowolny kod HTML, JavaScript lub CSS. Strona ze znacznikami Visualforce jest przetwarzana i renderowana po stronie serwera, pozostawiając klientowi obsługę bogatego interfejsu użytkownika, korzystającego np. z jQuery.

Dostosowanie aplikacji Salesforce za pomocą stron Visualforce może dotyczyć np. utworzenia wielokrokowych kreatorów, dodania podsumowań czy rozszerzenia wyświetlanych danych w listach.

PROCESY BIZNESOWE

Dostosowanie czy zbudowanie aplikacji na platformie Salesforce związane jest zawsze ze zdefiniowaniem przynajmniej jednego z poniższych procesów biznesowych.

Reguły walidacji służą do określenia poprawności danych wprowadzonych przez użytkownika. Są one zapisane przy użyciu języka wyrażeń będącego podzbiorem języka Apex. Walidacja wykonana jest po stronie serwera Salesforce po kliknięciu przez użytkownika przycisku zapisz (lub innego funkcjonalnie mu odpowiadającego). W każdym przypadku edytowany jest rekord należący do danego obiektu, ale reguły walidacyjne mogą odnosić się do obiektów powiązanych, dla przykładu, można wymóc, aby suma wartości składowych zamówienia nie przekroczyła pewnej kwoty przy płatności gotówką.
Użytkownik zostanie poinformowany stosownym komunikatem, zaś rekord nie zostanie zapisany do bazy danych.

Reguły przepływu służą do automatyzacji pewnych czynności lub powiadamiania o oczekiwanych zmianach. Składają z trzech elementów:

- ustawień, które decydują w jakich okolicznościach kryteria będą obliczane, np. czy rekord został utworzy czy został edytowany,
- kryteriów, które mówią czy dany przepływ ma być uruchomiony,
- działań natychmiastowych lub zależnych od czasu, jak wysłanie emaila czy zmiana wartości pola.

Za pomocą procesów zatwierdzania można zdefiniować procesy, które zależą od akceptacji pewnej liczby osób (użytkowników aplikacji Salesforce). Programista definiuje kroki niezbędne do zakceptowania rekordu, a w szczególności osoby lub grupy osób zatwierdzające rekord. Można zdefiniować kryteria dla każdego kroku jak również akcje możliwe do podjęcia po zatwierdzeniu czy odrzuceniu rekordu. Przykładowym zastosowaniem takiego procesu może być wniosek o urlop kierowany do bezpośredniego przełożonego albo wielokrokowy proces rekrutacyjny.

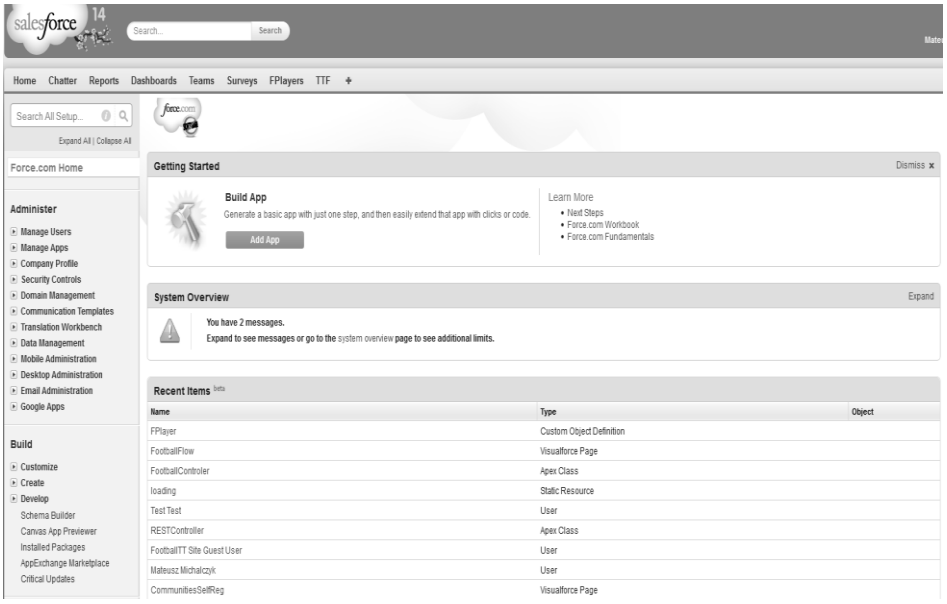
APEX CODE

Język Apex to silnie-typowalny język programowania na platformie Force.com. Można w nim definiować wyszukaną logikę biznesową, wyzwalacze bazodanowe oraz kontrollery interfejsu użytkownika. Składnia tego języka podobna jest do języka Java, choć sam język nie jest wrażliwy na wielkość liter. Jedną z najważniejszych tego języka jest podobieństwo do języków typu

PL/SQL, gdzie obiekty bazodanowe są obiektami pierwszej, tzn. nie musi zachodzić żadna jawna konwersja między wartościami zwracanymi przez zapytanie typu SELECT a obiektami języka programowania. Można definiować klasy i interfejsy, a oprócz standardowych elementów kontroli wykonania jak instrukcje warunkowe czy iteracyjne można również korzystać z wyjątków. Dostępne są wszystkie typy pól obiektów Salesforce (liczby, daty, napisy) a także kolekcje jak listy, mapy i zbiory.

NARZĘDZIA DLA SALESFORCE

Rozdział ten przedstawi wybrane narzędzia wspomagające proces tworzenia oprogramowania na platformie Force.com.

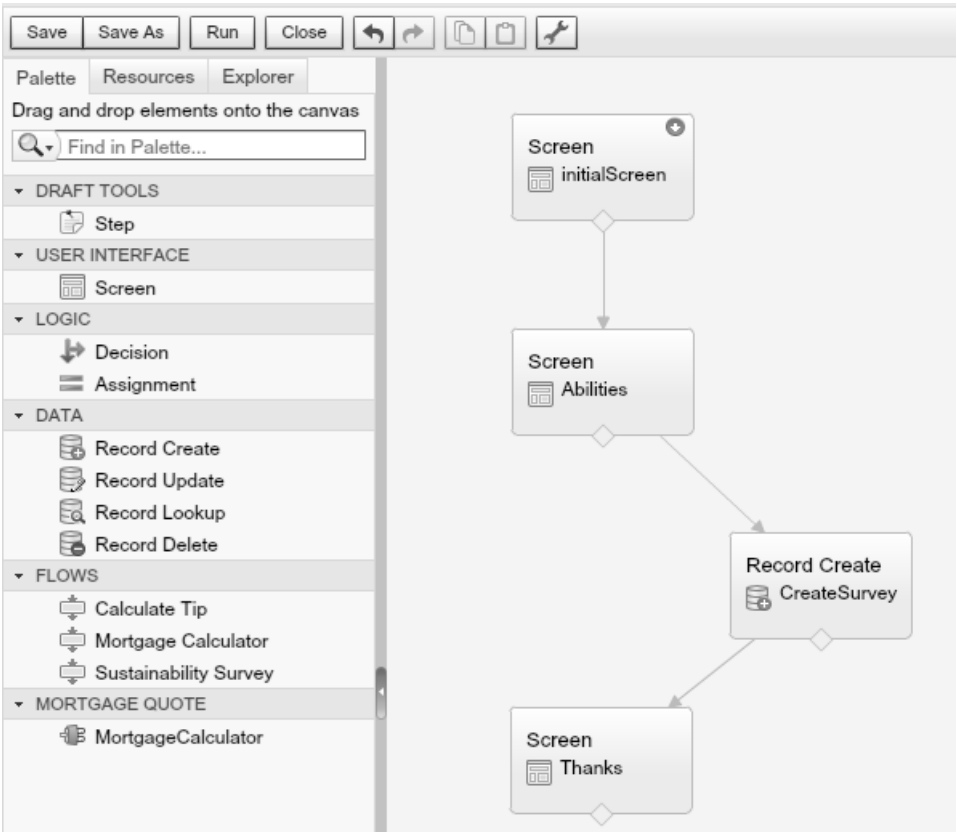


Rys.3. Platforma Force.com wersja Spring 14
(źródło: opracowanie własne)

Force.com (przedstawiony na Rys. 3) to platforma oferująca ekosystem wyposażony w narzędzia umożliwiające tworzenie aplikacji, nawet bez pisania natywnego kodu. Deweloper posiada możliwość tworzenia biznesowych aplikacji bazodanowych o wysokim poziomie abstrakcji obsługujących przepływy biznesowe, wysyłanie maili lub sms-ów oraz standardowe definicje obiektów.

Całość dostępna jest z poziomu przeglądarki. Standardowe aplikacje tworzone bez pisania kodu mogą posiadać interaktywne przepływy, walidacje oraz inne, często używane procesy biznesowe. Rys. 4 przedstawia tworzenie przykładowego przepływu.

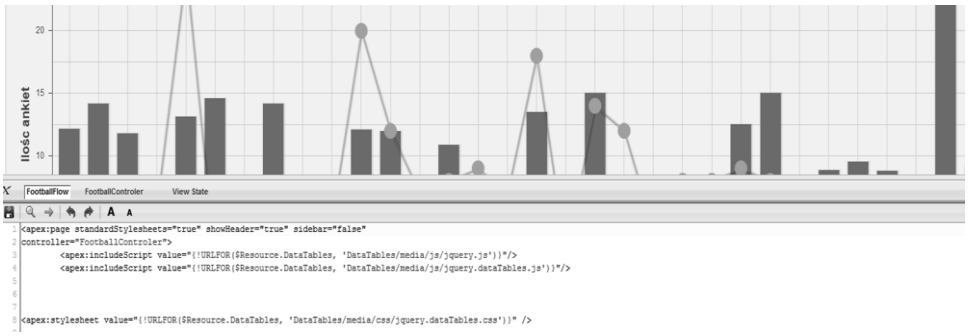
Ze względu na osadzenie platformy na chmurze, użytkownik nie musi się martwić problemami sprzętowymi, platforma powinna dbać o wszystko. Deweloperzy powinni skupić się na tworzeniu aplikacji. Sama platforma poza możliwością tworzenia aplikacji staje się również miejscem korzystania z tych aplikacji, tak więc zarówno deweloperzy, administratorzy, jak i końcowi klienci korzystają z tego samego narzędzia mają możliwość w łatwy sposób rozszerzyć ilość urządzeń, z których korzystają odpowiednio dostosowanego do ich potrzeb.



Rys. 4. Tworzenie przepływów
(źródło: opracowanie własne).

Dla aplikacji niestandardowych lub wymagających nieobsługiwanych funkcjonalności platforma Force.com dostarcza nam głównie dwa narzędzia, jest to język programowania Apex oraz język szablonów HTML o nazwie Visualforce.

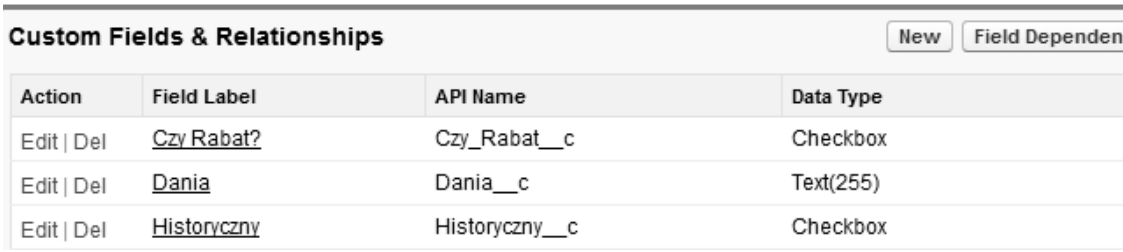
Obydwa języki dobrze ze sobą współpracują, a praca przy tworzeniu strony w przeglądarce wydaje się prosta i intuicyjna (Rysunek 5).



Rys. 5. Tworzenie aplikacji za pomocą VF oraz Apex
(źródło: opracowanie własne)

Tworzenie definicji obiektów jest przeprowadzone całkowicie deklaratywnie co pokazują Rysunki 6 oraz 7. Do wyboru mamy różne typy danych takie jak:

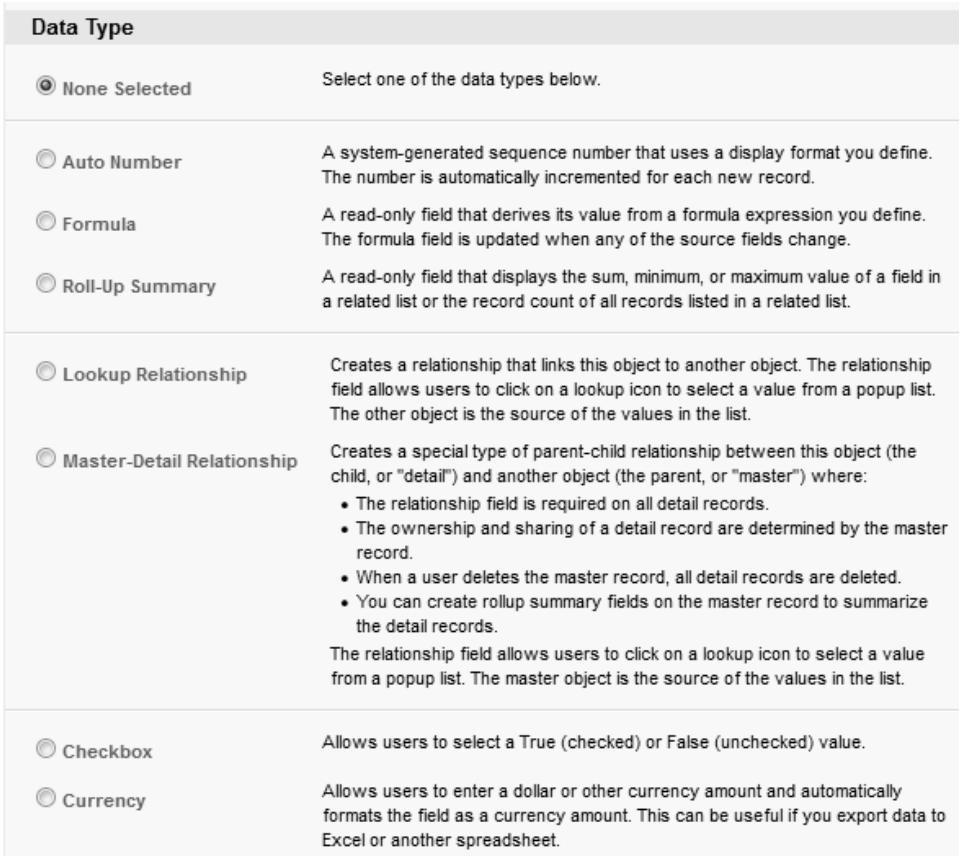
- Auto numer – numeracja automatyczna, można zdefiniować swój własny wzór numeracji.
- Formula– pole tylko do odczytu przedstawiające jakąś wartość na podstawie innych danych z tabeli lub tabel pochodnych. Formuły pisane są swego rodzaju pseudokodem.
- Email – pole walidowane na schemat adresu e-mail.
- Number– Pole numeryczne, może przedstawić również liczbę zmiennoprzecinkową.
- Text – Pole tekstowe.
- Master – Detail Relationship – Pozwala na stworzenie trwałej relacji Rodzic – Dzieci z inną tabelą.



Custom Fields & Relationships New | Field Dependen

Action	Field Label	API Name	Data Type
Edit | Del	Czy Rabat?	Czy_Rabat__c	Checkbox
Edit | Del	Dania	Dania__c	Text(255)
Edit | Del	Historyczny	Historyczny__c	Checkbox

Rys. 6. Definiowanie obiektów
(źródło: opracowanie własne)



Data Type

None Selected	Select one of the data types below.
Auto Number	A system-generated sequence number that uses a display format you define. The number is automatically incremented for each new record.
Formula	A read-only field that derives its value from a formula expression you define. The formula field is updated when any of the source fields change.
Roll-Up Summary	A read-only field that displays the sum, minimum, or maximum value of a field in a related list or the record count of all records listed in a related list.
Lookup Relationship	Creates a relationship that links this object to another object. The relationship field allows users to click on a lookup icon to select a value from a popup list. The other object is the source of the values in the list.
Master-Detail Relationship	Creates a special type of parent-child relationship between this object (the child, or "detail") and another object (the parent, or "master") where: • The relationship field is required on all detail records. • The ownership and sharing of a detail record are determined by the master record. • When a user deletes the master record, all detail records are deleted. • You can create rollup summary fields on the master record to summarize the detail records. The relationship field allows users to click on a lookup icon to select a value from a popup list. The master object is the source of the values in the list.
Checkbox	Allows users to select a True (checked) or False (unchecked) value.
Currency	Allows users to enter a dollar or other currency amount and automatically formats the field as a currency amount. This can be useful if you export data to Excel or another spreadsheet.

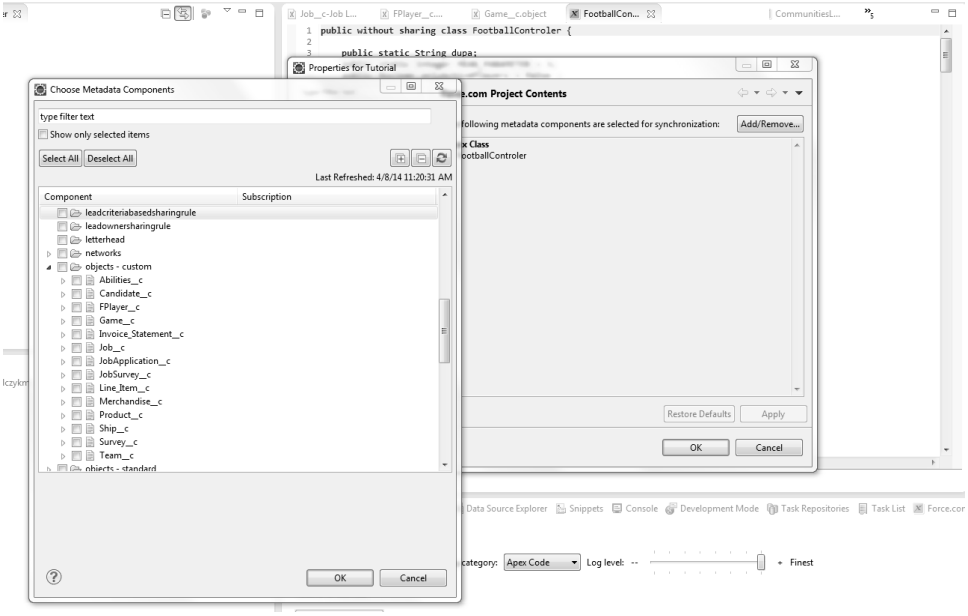
Rys. 7. Ekran wyboru typu danej
(źródło: opracowanie własne)

Ze względu na potrzebę tworzenia wyspecjalizowanych aplikacji często połączonych z innymi systemami Salesforce stworzył Developer Console, IDE dla języka Apex. Największą zaletą opisywanego IDE jest jego dostępność z poziomu przeglądarki oraz dużo lepsze podpowiedzi kontekstowe w porównaniu do konkurencyjnych IDE. Nie bez znaczenia jest również to iż to Salesforce.com jest twórcą Developer Console co za tym idzie, wsparcie oraz kompatybilność teoretycznie powinna być większa niż w przypadku innych IDE. Wszelki kod zapisany na platformie automatycznie staje się aktywny i będzie się wykonywał przy wywołaniu. Force.com nie pozwala na zapisywanie kodu z błędami składniowymi (niekompilującego się) jak i nie prowadzi kontroli wersji.
Przykładowy widok Developer Console przedstawia Rysunek 8.

Rys.8. Developer Console
(źródło: opracowanie własne)

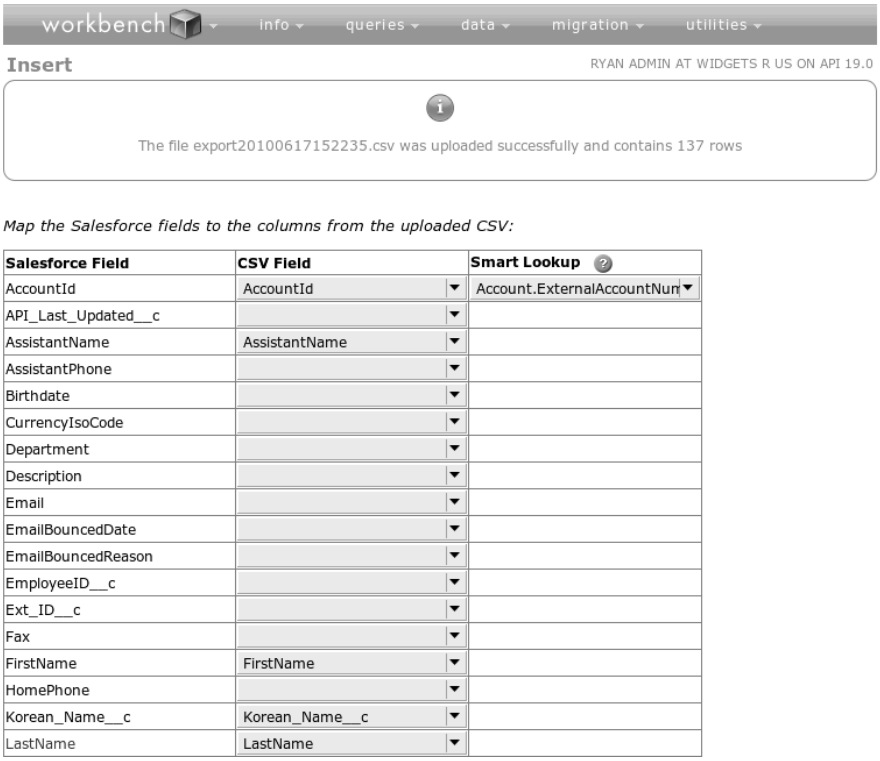
Force.com IDE (Rys. 9) jest pluginem do narzędzia Eclipse w wersjach 4.3 Kepler oraz 4.2 Juno. Dodatek oferuje wszystkie funkcjonalności potrzebne do wdrożenia obiektów przesyłanych poprzez Salesforce Metadata ,kontrolerów, wyzwalaczy [1]. IDE oferuje również wywoływanie anonimowe, które polega na wywołaniu kawałka kodu bezpośrednio z konsoli, bez potrzeby wdrażania tego kodu na platformę Force.com. Główną zaletą Force.com względem Developer Console jest możliwość pracy w zespole za pomocą systemu kontroli wersji np. Git lub SVN. Oferuje to niespotykaną wcześniej elastyczność, pracy na osobnych środowiskach platformy Force.com (ang. Sandbox) przy jednocześnie łatwym kanale synchronizacji kodu oraz obiektów na środowisko produkcyjne lub testowe. Oczywiście ważnym elementem tutaj jest również pełna funkcjonalność systemów kontroli wersji, zwłaszcza ze względu na to iż platforma Force.com nie wersjonuje kodu, obiektów, czy jakichkolwiek modeli.

Z wad Force.com IDE można by wymienić ograniczone podpowiedzi kontekstowe co zdecydowanie przekłada się na szybkość pisania aplikacji na platformę Force.com.



Rys. 9. Synchronizowanie projektu poprzez obiekty Metadata w Force.com IDE (źródło: opracowanie własne)

Ostatnim narzędziem przedstawionym w tej sekcji zostanie darmowa aplikacja sieciowa Workbench. Stworzona została ona do interakcji deweloperów oraz administratorów z ich środowiskami Salesforce poprzez Force.com API. Workbench posiada szerokie wsparcie dla wszystkich dostępnych API na platformie Force.com takich jak: REST, SOAP, BULK, Apex oraz Metadata, które pozwalają na definicje obiektów oraz migracje danych. Dodatkowo aplikacja posiada szereg funkcjonalności wspomagających testowanie oraz debugowanie dla aktualnej oraz poprzednich wersji środowiska Force.com. Korzystanie z aplikacji jest intuicyjne ze względu na możliwość bezpośredniego logowania się na swoje środowisko poprzez Workbench. Aplikacja jest szczególnie przydatna przy testowaniu wstecznej kompatybilności aplikacji [3].
Rysunek 10 przedstawia migracje danych z pliku CSV za pomocą narzędzia Workbench.



Rys.10. Mapowanie pól obiektów na kolumny pliku CSV w Workbench (źródło:[10])

PODSUMOWANIE

Rozwiązanie proponowane przez Salesforce są jednymi z najbardziej innowacyjnych co potwierdza ich pierwsze miejsce na liście „The World's Most Innovative Companies" [7]. Pierwsze miejsce na tej liście zajmują nieprzerwanie od 2011 roku [8].

Potwierdza to, iż biznes zmierza w kierunku chmur obliczeniowych. Zalety tych rozwiązań pozwalają na skupienie się na relacjach z klientem, marketingu, sprzedaży, bez obawy o infrastrukturę.

LITERATURA

[1] http://developer.salesforce.com/page/Force.com_IDE (dostęp 2014-05-30).
[2] *Salesforce.com Platform as a Service Review*, http://www.crmsearch.com/salesforce-paas-review.php (dostęp 2014-05-30).
[3] https://developer.salesforce.com/page/Workbench (dostęp 2014-05-30).
[4] Fowler M.: *Patterns of enterprise application architecture.* Addison-Wesley Professional, s. 330–331, 2002.
[5] Burbeck S.: *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)*.
[6] Schaeffer C.: *Salesforce.com Review—An Independent Assessment*, http://www.crmsearch.com/salesforce-review.php (dostęp 2014-05-30).
[7] *The World's Most Innovative Companies List* http://www.forbes.com/innovative-companies/list/ (dostęp 2014-05-30).
[8] *Three Years In A Row*, http://www.salesforce.com/company/awards/most-innovative-companies-salesforce-no1-forbes.jsp (dostęp 2014-05-30).
[9] http://www.chades.net/ (dostęp 2014-05-30).
[10] https://developer.salesforce.com/page/File:Wb2_insert_smart_lookup.png (dostęp 2014-05-30).

KAMILA NOWAK[1], BEATA PAŃCZYK[2]

HTML5 W PRZEGLĄDARKACH MOBILNYCH

WSTĘP

U podstaw tworzenia klasycznych aplikacji internetowych leży język HTML, u podstaw tworzenia mobilnych aplikacji internetowych leży jego nowa wersja HTML5. Sam język HTML zawiera w sobie określone komponenty, takie jak znaczniki wraz z atrybutami, typy danych, referencje znakowe, odwołania, deklaracje typu dokumentu.

Głównym założeniem standardu HTML5 jest jego kompatybilność wsteczna umożliwiająca współpracę z całym otoczeniem starej wersji HTML4 i jej wszystkimi komponentami. HTML5 koryguje, pojawiające się w specyfikacji HTML 4, niejasności, dotyczące kwestii obsługi błędów, które stanowią jedną z podstawowych przyczyn, dla której wiele serwisów internetowych, napisanych z naruszeniem specyfikacji, w różnych przeglądarkach internetowych działa w niektórych poprawnie, w innych nie. Najbardziej zauważalne zmiany HTML5 w stosunku do jego poprzednika dotyczą semantyki piątej wersji języka [1].

Dzięki nowym znacznikom zwiększyła się przejrzystość struktury dokumentów i komfort zarządzania formularzami, zadbano o dodatkowe funkcje do geolokalizacji, obsługi zawartości multimedialnej i bezpośredniego konstruowania grafiki w treści strony. HTML5 pozwala również wykorzystać możliwości nowoczesnych przeglądarek wbudowanych w urządzenia mobilne. Twórcy stron mogą korzystać z tego samego zestawu rozwiązań przy tworzeniu aplikacji i witryn WWW dla komórek i tabletów, które stosują przy budowaniu aplikacji internetowych dla komputerów stacjonarnych i laptopów.

Niestety jeszcze nie wszystkie możliwości HTML5 są obsługiwane w przeglądarkach mobilnych, dlatego ważne jest wskazanie elementów, które można bezpiecznie stosować projektując swoje aplikacje WWW oraz te, z których na razie nie należy korzystać.

[1] Politechnika Lubelska, Wydział Elektrotechniki i Informatyki,
studentka kierunku Informatyka
[2] Politechnika Lubelska, Wydział Elektrotechniki i Informatyki, Instytut Informatyki

WERSJE MOBILNE POPULARNYCH PRZEGLĄDAREK

Rozwój przeglądarek mobilnych nastąpił wraz z wkroczeniem smartfonów na rynek telefonów komórkowych. Do tej pory, w większości posiadacze tych urządzeń aktywnie korzystali z Internetu w tradycyjny sposób i mają swoje przyzwyczajenia jeśli chodzi o przeglądarkę. Dlatego, pomimo dostępności automatycznie wbudowanych w smartfony aplikacji do przeglądania Internetu, wybierają mobilne odpowiedniki tych znanych z komputerów osobistych.

Firefox for Mobile to mobilna wersja przeglądarki autorstwa Mozilli dostępna dla urządzeń z systemem Android, niestety nie skorzystają z niej posiadacze urządzeń z systemem iOS. Bazuje na tej samej technologii co jej desktopowy odpowiednik. Posiada też wiele z jego funkcjonalności (Rys. 1).

Rys. 1. Firefox for Mobile na urządzeniu mobilnym (źródło:[2])

Mobilny Firefox jest przystosowany do wykonywania poleceń przekazywanych za pomocą gestów np. powiększanie lub pomniejszanie strony poprzez dwukrotne stuknięcie w ekran lub wykonanie gestu zsunięcia palców. Urządzenia, na których zainstalowana jest wersja Androida 4.1 Jelly Bean lub nowsza,

pozwalają szybko przełączać się pomiędzy kartami najczęściej odwiedzanych stron, zakładek i historii, przewijając ekran w lewo i w prawo. Niewątpliwą zaletą jest wprowadzenie inteligentnego paska adresów (Rys. 2), który grupuje funkcję wybierania adresu, generowania podpowiedzi, wyszukiwania oraz przeglądania zakładek, historii i kart.

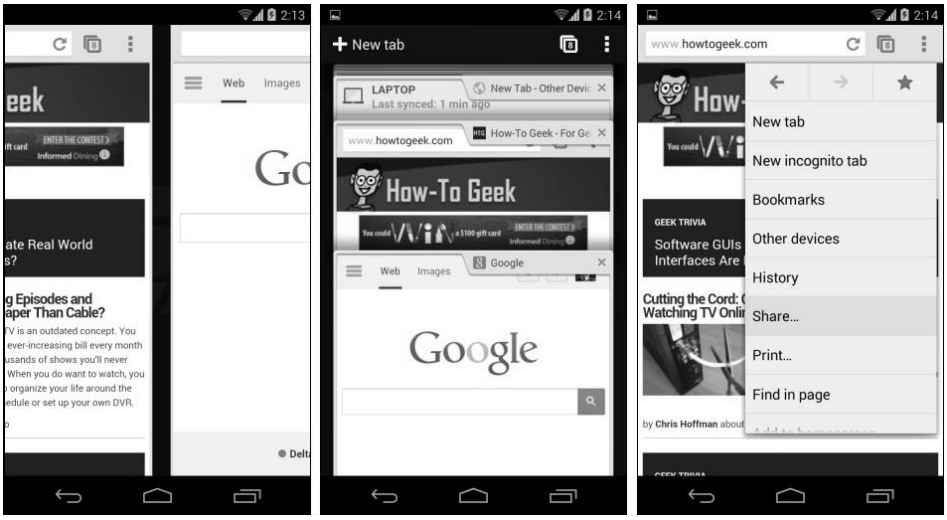
Rys. 2. Inteligentny pasek adresów Firefox Mobile
(źródło: [3])

Chrome for Mobile jest dostępna zarówno dla użytkowników Androida jak i systemu iOS. Wykorzystuje konto Google co pozwala na synchronizację otwartych kart, zakładek i historii między komputerem a telefonem lub tabletem. Oferuje możliwość jednoczesnego korzystania z wielu kart i ułatwia obsługę za pomocą gestów (Rys. 3) [4].

W przypadku smartfonów z systemem Android oraz urządzeń typu iPad i iPhone, po przesunięciu:

- horyzontalnym na pasku narzędzi można szybko zmieniać zakładki;
- w dół na pasku narzędzi można otworzyć widok zakładek;
- w dół na przycisku menu można otworzyć odpowiednią listę i wybrać interesującą funkcję bez odrywania palca od ekranu.

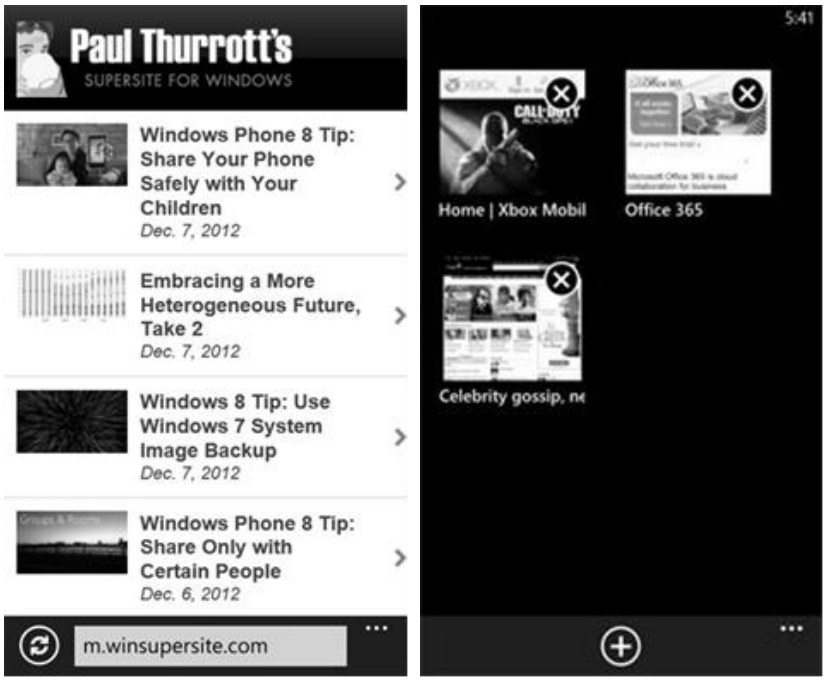
Na urządzeniu mobilnym, przeglądarka ta umożliwia swobodny dostęp do usługi wyszukiwania głosowego – Google Voice Search.



Rys.3. Obsługa gestów przeglądarki Chrome na urządzeniu mobilnym

Internet Explorer jest przeznaczony na urządzenia mobilne działające w oparciu o system operacyjny programistów Microsoft. Wersja mobilna, podobnie jak inne produkty z rodziny Internet Explorer na komputery stacjonarne i laptopy bazuje na silniku Trident. Pierwsze wydania udostępnione zostały pod nazwą Pocket Internet Explorer (w skrócie PIE). Przeglądarki mobilne używane na platformie Windows Mobile przez prawie dziesięć lat miały wyspecjalizowany mobilny silnik przetwarzania układu. Ostatnia wersja przeglądarek z tej rodziny, udostępniona mniej więcej w 2002 roku, działała na urządzeniach z systemem Windows Mobile do 2008 roku [5]. Mimo niedoskonałego przetwarzania układu stron, nadal jest używana.

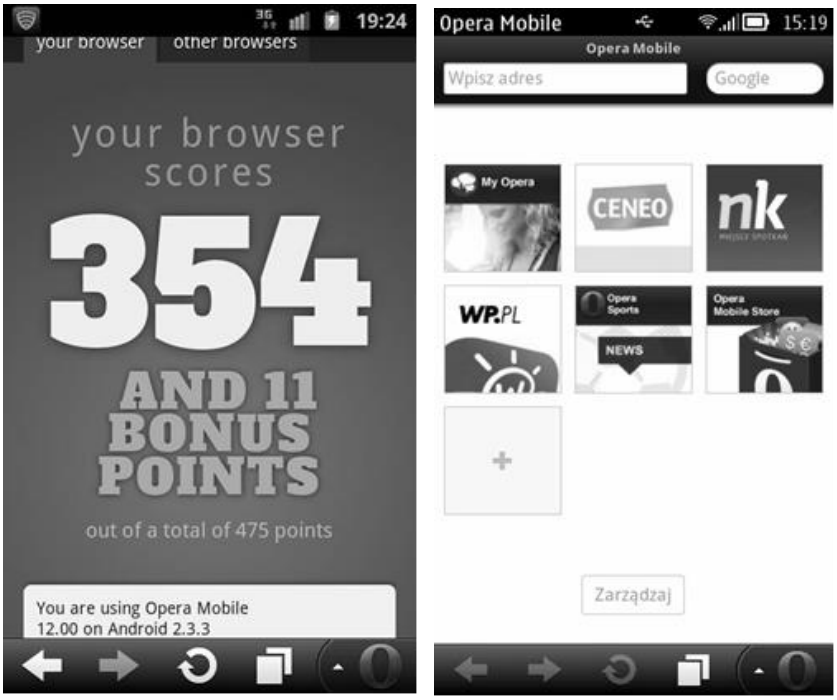
Wraz z pojawieniem się nowej wersji mobilnego systemu operacyjnego od Microsoft – Windows Phone 7, zmiany objęły także i przeglądarkę. W lutym 2011 roku na Mobile World Congress, zaprezentowana została wersja 9 programu (Rys. 4). Ważny element prezentacji stanowił pokaz możliwości interpretacji znaczników HTML5 przez najnowsze wydanie mobilnego Internet Explorer.



Rys. 4. Strona internetowa i obsługa zakładek w IE na urządzeniu mobilnym (źródło: [6, 7])

Opera jest również dostępna dla urządzeń przenośnych. Na każdym z systemów zapewnia ten sam silnik renderujący kod HTML. W przypadku urządzeń małoekranowych przeglądarka wyposażona jest o mechanizm zwany *small screen rendering*, który pozwala oglądać szerokokątne i wzbogacone w grafiki strony internetowe na wyświetlaczach uboższych i mniejszych niż ekrany „pełnowymiarowych" komputerów.

W wersji dla telefonów komórkowych Opera dostępna jest w dwóch bezpłatnych wersjach: Opera Mobile (Rys.5) dla urządzeń z systemem Symbian, Windows Mobile lub Android, oraz Opera Mini [8] dla urządzeń z systemem Apple iOS oraz dla każdego telefonu komórkowego z zainstalowanym klientem Java opartym na CLDC.



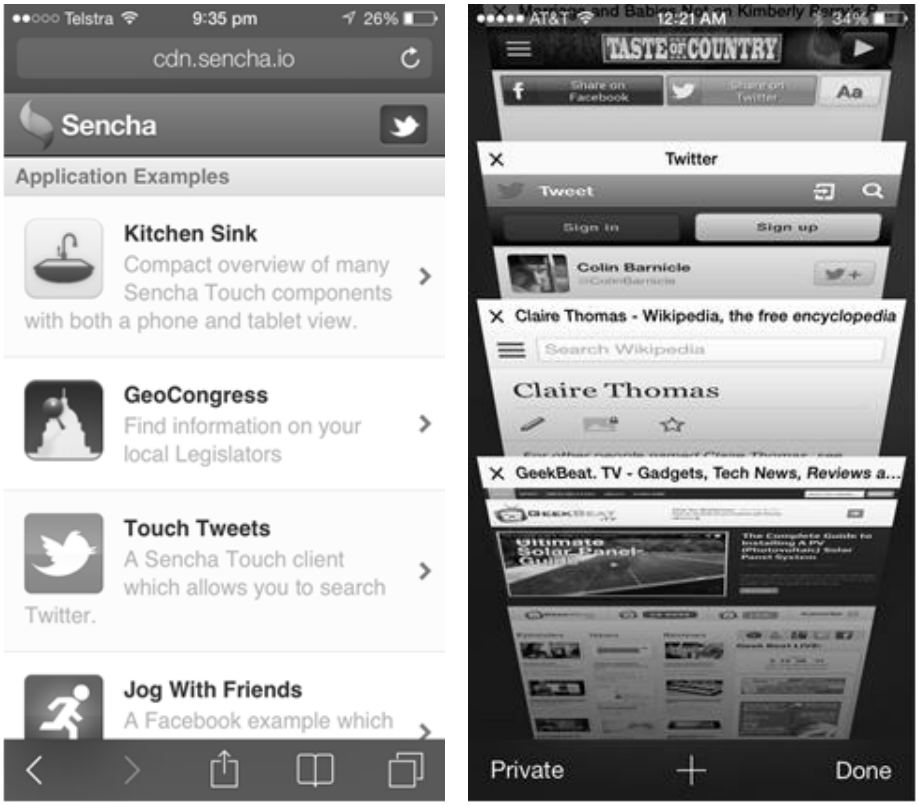
Rys. 5. Przeglądarka Opera Mobile 12 na urządzeniu mobilnym (źródło: [9])

Wersja Mini po uruchomieniu i wybraniu adresu strony nawiązuje połączenie z serwerem Opery, który dokonuje konwersji tradycyjnej strony WWW tak, by mogła być ona wyświetlona na ekranie urządzenia przenośnego (zwykle jest to telefon komórkowy) (Rys. 6.). Strony przed wysyłaniem (transmisja jest szyfrowana algorytmem RC4 z 256-bitowym kluczem a wymiana kluczy symetrycznych zabezpieczona algorytmem RSA z 1280-bitowym kluczem) z serwera do przeglądarki są kompresowane, co przyspiesza ich ładowanie w telefonie i znacznie ogranicza ilość transferowanych danych.

Opera Mini nie jest dobrym wyborem jeśli chodzi o przeglądanie zabezpieczonych stron internetowych. Ze względu na to, że wszystkie witryny są najpierw przekierowywane i ładowane za pośrednictwem serwerów Opery, taka strona, przed wysłaniem z powrotem do przeglądarki, będzie już wcześniej odszyfrowana i ponownie zaszyfrowana.

Takie witryny lepiej obsłuży wersja Mobile. Zdecydowanie też lepiej niż Mini radzi sobie z renderowaniem stron, tak, że wyglądają one dokładnie jak na komputerach PC.

Przeglądarka **Safari** w wariancie przenośnym to **Safari Mobile** (Rys. 6). Została wprowadzona na rynek latem 2007 i stanowiła oprogramowanie wbudowane w telefon komórkowy iPhone i odtwarzacz MP3 iPod.



Rys. 6. Strona internetowa z obsługą zakładek w Safari na urządzeniu mobilnym (źródło:[10,11])

Safari dla urządzeń mobilnych jako pierwsza, pozwalała na wyświetlanie na urządzeniu przenośnym stron WWW w sposób identyczny jak na zwykłym komputerze, nie tracąc przy tym na funkcjonalności. Bazująca na silniku WebKit, wyposażona jest w szereg dodatkowych funkcjonalności co wyprzedza konkurencję w testach wydajności i czyni ją liderem w rankingach popularności mobilnych przeglądarek.

WZORCE TESTOWE

W celu przeprowadzenia analizy obsługi elementów HTML5 w przeglądarkach mobilnych zostały sporządzone wzorce testowe, spełniające reguły walidatora W3C [12].

Wykorzystano wzorce:

- dla znaczników struktury dokumentu,
- dla znacznika *canvas*,
- dla znaczników obsługujących multimedia,
- dla znaczników obsługujących formularze.

Przeprowadzone testy miały na celu stwierdzenie czy HTML5 (w momencie sprawdzania) może zostać uznany za standard tworzenia współczesnych stron i aplikacji internetowych na urządzenia mobilne. W badaniu wykorzystane zostały przykładowe wzorce sporządzone zgodnie z regułami piątej wersji HTML. Na ich podstawie sformułowano wnioski odnośnie stopnia poprawnej interpretacji wybranych elementów HTML5 w pięciu mobilnych odpowiednikach popularnych przeglądarek. Do opracowania wykorzystano również dane uzyskane z serwisów internetowych gromadzących informacje o HTML5, takich jak:

- html5test.com [13],
- html5demos.com [14],
- caniuse.com [15],
- mobilehtml5.org [16],

oraz oficjalną stronę organizacji zajmującej się ustanawianiem standardów WWW [17].

Dodatkowo wykorzystana została biblioteka w języku JavaScript: Modernizr [18]. Biblioteka dostarcza rozwiązań pozwalających na detekcję wybranych składników HTML5 w dowolnej przeglądarce.

Przykład wykorzystania tej biblioteki do sprawdzenia możliwości obsługi danego znacznika przez przeglądarki internetowe przedstawia Listing 1.

Listing 1.Sprawdzenie wybranego elementu HTML5 za pomocą biblioteki Modernizr

```
1   <!DOCTYPE html>
2   <html lang="en" class="no-js">
3   <head>
4     <meta charset="utf-8">
5     <title>Modernizr test: week</title>
6     <script type="text/javascript"
                src="js/modernizr-zn-test.js">
7     </script >
8   </head>
9   <body>
10   <div id="test">
11     <div id="test-week">test </div>
12   </div>
13   <script>
14     if (!Modernizr.inputtypes.week) {
15      document.getElementById('test-week').innerHTML
                  ='Typ week nie jest obsługiwany';
16     } else
17       document.getElementById('test-week').innerHTML
                  ='Przegladarka obsługuje typ: week';
18   </script>
19  </body>
20  </html>
```

Sprawdzenia kompatybilności znaczników HTML5 dokonano w oparciu o wybrane przeglądarki działające na różnych systemach mobilnych:

- Mozilla Firefox dla systemu Android;
- Google Chrome dla systemu Android;
- Internet Explorer Mobile;
- Safari Mobile dla systemu iOS7,
- Operę w wersji Mobile.

WYNIKI

Tabele 1-4 przedstawiają kompatybilność znaczników standardu HTML5 z przeglądarkami urządzeń mobilnych.

Tabela 1.Interpretacja znaczników struktury dokumentu HTML5 w przeglądarkach mobilnych (opracowanie własne)

Znacznik	Firefox	Chrome	IE Mobile	Opera Mbile	Safari
<header>	Tak	Tak	Tak	Tak	Tak
<footer>	Tak	Tak	Tak	Tak	Tak
<section>	Tak	Tak	Tak	Tak	Tak
<article>	Tak	Tak	Tak	Tak	Tak
<aside>	Tak	Tak	Tak	Tak	Tak
<details>	Nie	Tak	Nie	Nie	Tak
<main>	Tak	Tak	Nie	Nie	Tak
<dialog>	Nie	Nie	Nie	Nie	Nie
<summary>	Nie	Tak	Nie	Nie	Tak

Tabela 2. Interpretacja znaczników HTML5 do obsługi multimediów(opracowanie własne)

Znacznik	Firefox	Chrome	IE Mobile	Opera Mobile	Safari
<audio>	Tak	Tak	Tak	Tak	Tak
<video>	Tak	Tak	Tak	Tak	Tak
<source>	Tak	Tak	Tak	Tak	Tak
<track>	Nie	Tak	Nie	Nie	Tak

Tabela 3. Kompatybilność znaczników HTML5 dla obrazów i ich atrybutów (opracowanie własne)

Znaczniki i ich atrybuty	**Firefox**	**Chrome**	**IE Mobile**	**Opera Mobile**	**Safari**
<img>					
crossorgin	Tak	Tak	Tak	Tak	Tak
<map> gdy ustawiony jest atrybut *id,* <map> wymaga tej samej wartości co atrybut *name*	Tak	Tak	Tak	Tak	Tak
<area>					
download	Tak	Tak	Tak	Tak	Tak
media	Tak	Tak	Tak	Tak	Tak
hreflang	Tak	Tak	Tak	Tak	Tak
rel	Tak	Tak	Tak	Tak	Tak
type	Tak	Tak	Tak	Tak	Tak
<canvas>	Tak	Tak	Tak	Tak	Tak
<figcaption>	Tak	Tak	Tak	Tak	Tak
<figure>	Tak	Tak	Tak	Tak	Tak

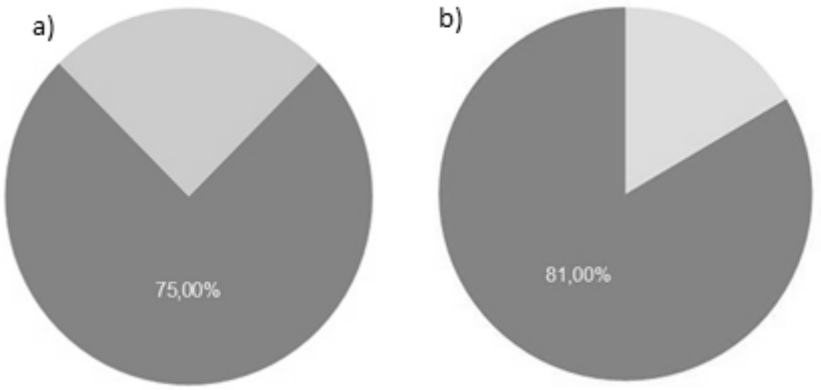
W Tabeli 3 zastosowano następujące oznaczenia:

- *atrybut* – nowe atrybuty dla istniejących znaczników
- **<element>** – elementy dostarczone wraz z HTML5

Tabela 4. Kompatybilność nowych typów pól formularza w przeglądarkach mobilnych (opracowanie własne)

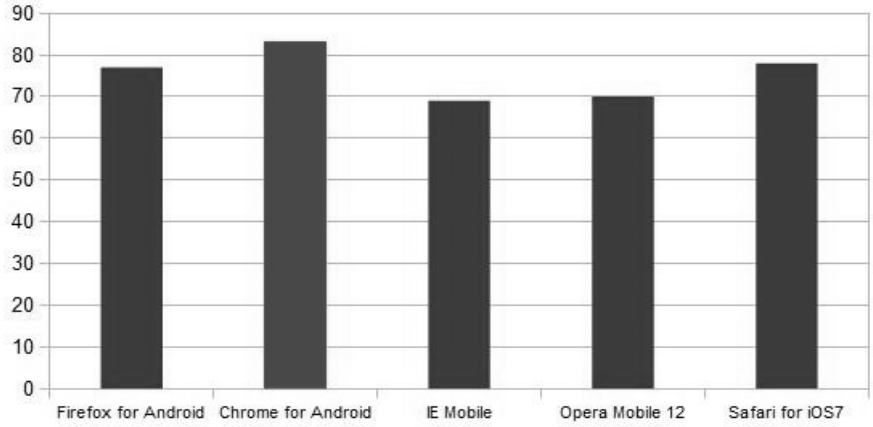
Znacznik	Firefox	Chrome	IE Mobile	Opera Mobile	Safari
<input type="search">	Tak	Tak	Tak	Tak	Tak
<input type="range">	Tak	Tak	Tak	Tak	Tak
<input type="number">	Tak	Tak	Tak	Tak	Tak
<input type="color">	Nie	Tak	Nie	Tak	Nie
<input type="url">	Tak	Tak	Tak	Tak	Tak
<input type="email">	Tak	Tak	Tak	Tak	Tak
<input type="tel">	Tak	Tak	Tak	Tak	Tak
<input type="date">	Tak	Tak	Nie	Tak	Nie
<input type="month">	Nie	Tak	Nie	Tak	Nie
<input type="week">	Nie	Tak	Nie	Tak	Nie
<input type="time">	Tak	Tak	Nie	Tak	Nie
<input type="datetime">	Nie	Nie	Nie	Tak	Nie
<input type="datetime-local">	Nie	Tak	Nie	Tak	Nie

Analiza przeglądarek mobilnych wykazała, że ich skuteczność w interpretacji znaczników HTML5 wynosi 75% . Dla porównania przeglądarki desktopowe interpretują elementy HTML5 w 81% (Rys.7) [19].

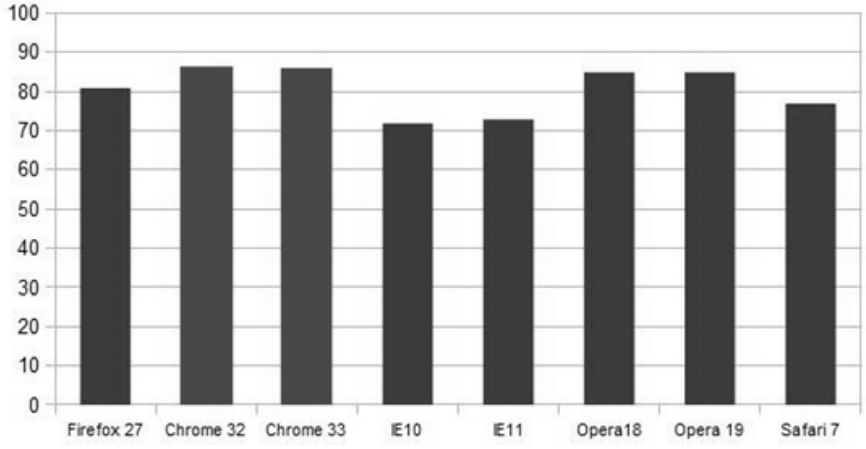


Rys 7. Średni wynik skuteczności interpretacji HTML5 dla przeglądarek na urządzeniach a) mobilnych, b) desktopowych
(źródło: opracowanie własne)

Największy stopień kompatybilności wykazała, tak jak w przypadku wersji desktopowej (Rys. 9) [19], przeglądarka Google Chrome (Rys. 8), w 83% poprawnie rozpoznając znaczniki HTML5.



Rys 8. Obsługa znaczników HTML5 w wybranych przeglądarkach mobilnych
(źródło: opracowanie własne)



Rys 9. Obsługa znaczników HTML5 w wybranych przeglądarkach desktopowych (źródło: opracowanie własne)

Mobilne urządzenia wyposażone w analizowane przeglądarki są w stanie prawidłowo rozróżnić elementy z grupy odpowiadającej za podstawową strukturę dokumentu w nowej specyfikacji (stopka, nagłówek, sekcja itd.).

Nowe pola i atrybuty dla formularzy udostępniają szereg różnorodnych typów pól, pozwalając na wprowadzanie danych tylko o określonym formacie. Grupa jest w całości poprawnie interpretowana w przeglądarce Opera Mobile 12. W przypadku Chrome wynik jest zbliżony i wynosi 93%. Zdolność do rozpoznawania nowych typów pól przez pozostałe analizowane programy zawiera się między 46% a 62%.

Stosunkowo najgorzej z obsługą nowych znaczników radzi sobie Internet Explorer, który w wersji na urządzenia mobilne poprawnie interpretuje mniej niż 70% elementów.

Przeprowadzone testy pozwoliły wyodrębnić znaczniki HTML5, które we wszystkich przeglądarkach zostały pomyślnie zweryfikowane i takie, których nie był w stanie rozpoznać żaden z wybranych programów do przeglądania Internetu (Tabela 5).

Tabela 5. Obsługa znaczników HTML5 – podsumowanie (opracowanie własne)

Wybrane przeglądarki mobilne – obsługa znaczników HTML5	
pełna	**brak**
<hidden > <data> <contextmenu >	<translate>
<input type="url"> <input type="email">	<contenteditable >
<input type="tel"> <input type="search">	<input type="datetime">
<input type="range"> <input type="number">	<rp>
<mark> <progress >	<rt>
<figcaption> <figure> <canvas> <source>	<menu>
<nav> <embed> <header> <footer>	<command>
<section> <article> <aside>	<dialog>

PODSUMOWANIE

HTML5 ewoluując w kierunku funkcjonalnych i interaktywnych stron WWW, wprowadza elementy pozwalające na budowanie w Internecie aplikacji na urządzenia mobilne, które oferowanymi funkcjonalnościami coraz bardziej upodabniają się do aplikacji natywnych.

Przeprowadzona analiza wskazuje, że najnowsze wersje popularnych przeglądarek mobilnych w dużej mierze interpretują nowe znaczniki i atrybuty HTML5. Ilość wdrożonych znaczników w porównaniu z niewielką liczbą elementów HTML5 zupełnie nie wspieranych przez współczesne przeglądarki i zarazem dostarczających funkcjonalności o marginalnym znaczeniu dla ogółu dokumentu, wskazuje, że projekty aplikacji mobilnych tworzone w oparciu o piąte wydanie HTML, można traktować jako bezpieczny standard.

W dużej mierze prawidłowość obsługi elementów i atrybutów HTML5 zależy od wyboru właściwej przeglądarki.

LITERATURA

[1] MacDonald M.: *HTML5. Nieoficjalny podręcznik*, O'Reilly, 2013.
[2] http://planet.firefox.com/mobile (dostęp 15.02.2014).
[3] http://support.mozilla.org/pl/kb/jak-uzywac-inteligentnego-paska-adresu (dostęp 15.02.2014).
[4] http://chrome.blogspot.com/2013/08/searching-by-image-gets-easier.html (dostęp 15.02.2014).
[5] Pearce J.: *Programowanie mobilnych stron internetowych z wykorzystaniem systemów CMS*, Helion 2012.
[6] http://winsupersite.com/windows-phone/internet-explorer-10-mobile-windows-phone-8 (dostęp 15.02.2014).
[7] http://www.windowsphone.com/en-US/How-to/wp8/web/use-internet-explorer (dostęp 15.02.2014).
[8] http://www.operasoftware.com/press/releases/mobile/opera-mini-drives-social-networking-on- mobile-phones (dostęp 15.02.2014).
[9] http://www.opera.com/pl/mobile (dostęp 15.02.2014).
[10] http://www.gizmag.com/icab-mobile-web-browser-iphone-ipad/14475/ (dostęp 15.02.2014).
[11] http://geekbeat.tv/review-ios-7 (dostęp 15.02.2014).
[12] http://validator.w3.org (dostęp 15.02.2014).
[13] http://html5test.com (dostęp 15.02.2014).
[14] http://html5demos.com (dostęp 15.02.2014).
[15] http://caniuse.com (dostęp 15.02.2014).
[16] http://mobilehtml5.org (dostęp 15.02.2014).
[17] http://www.w3.org/TR/html5 (dostęp 15.02.2014).
[18] http://modernizr.com (dostęp 15.02.2014).
[19] Nowak K.: *Analiza współczesnych przeglądarek pod kątem interpretacji znaczników HTML5*. Praca magisterska, Politechnika Lubelska, 2014.

STANISŁAW SKULIMOWSKI[1]

TWORZENIE GIER NA PLATFORMY MOBILNE Z WYKORZYSTANIEM FRAMEWORKA BOX2D

WSTĘP

Pomimo powtarzającego się głosu sprzeciwu wśród tradycjonalistów, rynek gier mobilnych wciąż rośnie, a możliwość użycia swojego tabletu lub telefonu jako konsoli do gier, staje się coraz częściej wykorzystywana.
Gry przeznaczone na te urządzenia już dawno przestały być prostymi migawkami i zgadywankami, które mogły zaciekawić odbiorcę najwyżej na kilka minut. Obecnie te gry stoją na tak wysokim poziomie, że śmiało mogą dorównywać popularności i jakości wykonania tytułów przeznaczonych na platformy tradycyjne, takie jak konsole stacjonarne czy komputery osobiste.

Możliwość wejścia na rynek gier mobilnych jest o tyle przyciągająca, że obecnie daje się zauważyć wciąż malejącą barierę wejścia od strony technologii. Oznacza to, że o ile wciąż pojawiają się narzędzia ułatwiające wytworzenie gry, o tyle coraz ciężej jest zdobyć dla niej odbiorców. Nie oznacza to jednak że jest to nie możliwe, a jedynie świadczy o konieczności wykazania się przez dewelopera większą kreatywnością.

Celem pracy było wykonanie gry możliwej do uruchomienia na urządzeniach mobilnych, korzystającej z funkcji biblioteki Box2D do emulowania zjawisk fizycznych i przeprowadzenie analizy wydajności.

Podstawą analizy będą informacje zawarte w dokumentacji, przekazywane na różne sposoby przez Internet oraz uzyskane od osób, które zgodziły się przetestować działanie aplikacji na swoich urządzeniach.

[1] Politechnika Lubelska, Wydział Elektrotechniki i Informatyki, Instytut Informatyki

ŚRODOWISKO PRACY

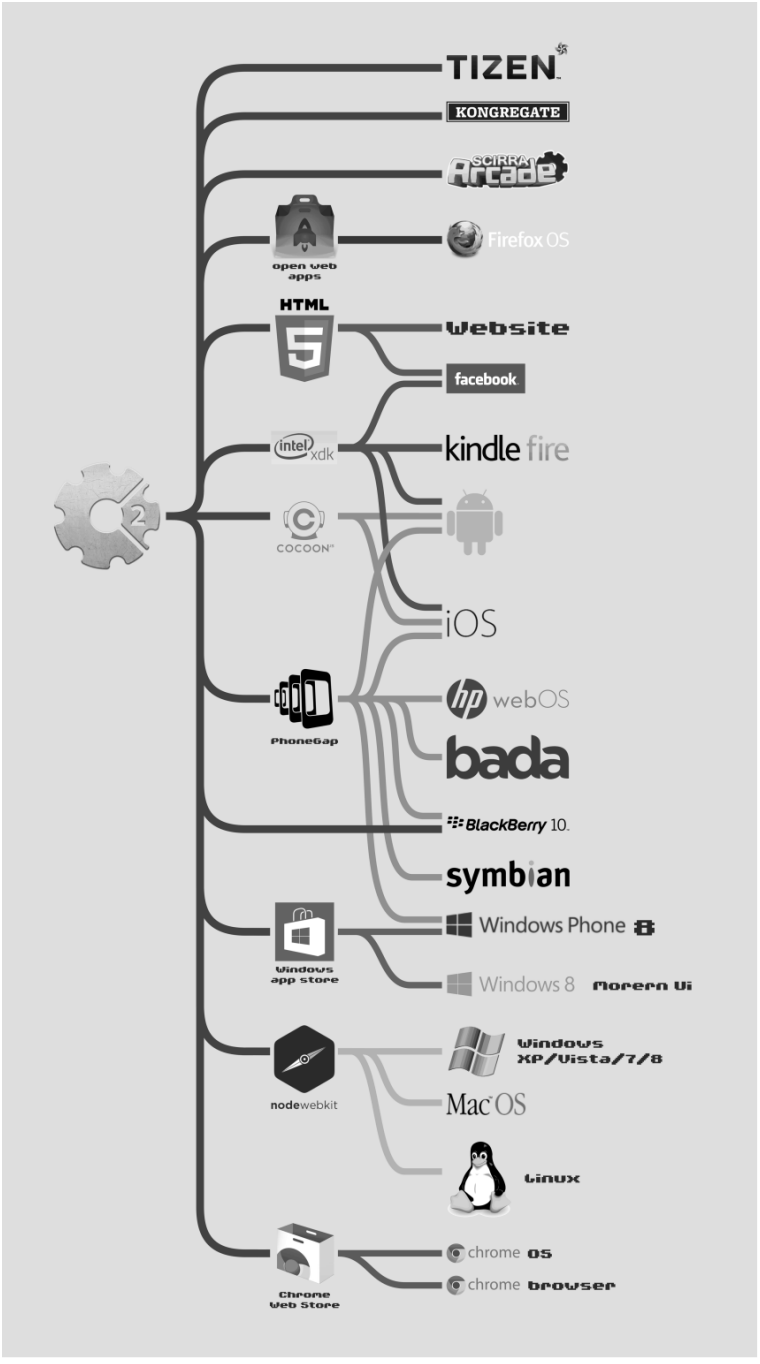
Do wytworzenia gry wykorzystano:

- Box2D – silnik służący do tworzenia symulacji zdarzeń fizycznych w przestrzeni dwuwymiarowej [1];
- Construct 2 – narzędzie do tworzenia gier, które domyślnie zapisuje je w formie kodu HTML5 i JavaScript [2];
- PhotoShop – program do edycji grafiki 2D;
- Toon Boom Studio – program do tworzenia animacji 2D;
- optiPNG – program do przepakowywania plików PNG do mniejszych struktur, wykorzystywany w celach optymalizacyjnych.

Rys. 1. Baner promocyjny programu Construct 2 na serwisie Steam (źródło: opracowanie własne)

Całość wykonania flagowego produktu firmy Scirra, stwarza u odbiorcy wrażenie, że ma do czynienia z produktem o wysokiej jakości. Świadczyć o tym mogą między innymi:

- dopracowany interfejs,
- rozbudowana dokumentacja,
- możliwość eksportowania projektów na wiele platform, rozdzielenie warstwy graficznej od warstwy programistycznej,
- możliwość samodzielnego rozbudowywania zakresu funkcjonalności



Rys. 2. Wykaz dostępnych dróg eksportu projektu wykonanego w programie Construct 2 (źródło: opracowanie własne)

FABUŁA

Akcja gry jest osadzona w przyszłości, około roku 2089, kiedy to rządy światowe zostały już przejęte przez korporacje. W tym świecie prawo stanowią najwięksi udziałowcy spółek i najbardziej rozpoznawalne marki, które opłacają wszystkie ważniejsze media i pracowników służb publicznych. Szerzące się epidemie, niedobór wody pitnej i pożywienia powodują wzrost napięcia społecznego. Przetrwanie gwarantuje życie w megametropoliach, za które obywatele muszą płacić swoją pracą dla korporacji. Często dochodzi do sytuacji, w których obywatele są traktowani jak zwierzęta lub gorzej – gdy nie są w stanie osiągnąć wystarczających wymagań, są eliminowani z życia społecznego, odbiera im się racje żywieniowe i wszelkie świadczenia na leki i opiekę zdrowotną, co przekłada się na powolną śmierć.

Bohaterem gry jest jeden z takich obywateli, który pomimo sumiennie wykonywanych obowiązków służbowych, stracił cały dobytek i wszelkie przywileje wynikające z tożsamości. W wyniku błędu systemu, został wykluczony z systemu. Jako były agent służb specjalnych, podjął czynną walkę z ciemiężycielami, jednakowoż żadna z przeprowadzanych przez niego akcji z użyciem brutalnej siły nie zakończyła się sukcesem.

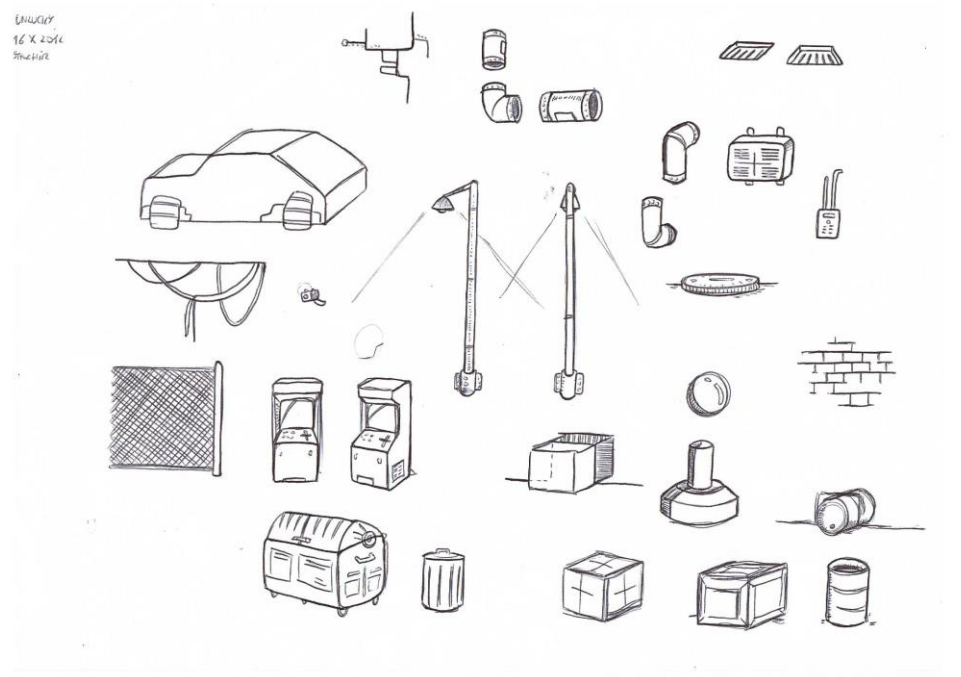
Dopiero działania w ukryciu, a często w tajemnicy, zaczęły przynosić efekty. Człowiek, któremu rzadko udaje się wygrać, postanawia nie poddać się, mimo że szczęście mu nie sprzyja. Dlatego właśnie przyjął imię Unlucky, w wolnym tłumaczeniu znaczące „feralny", „pechowy" lub „pozbawiony szczęścia".

Rys. 3: Projekt logo dla gry

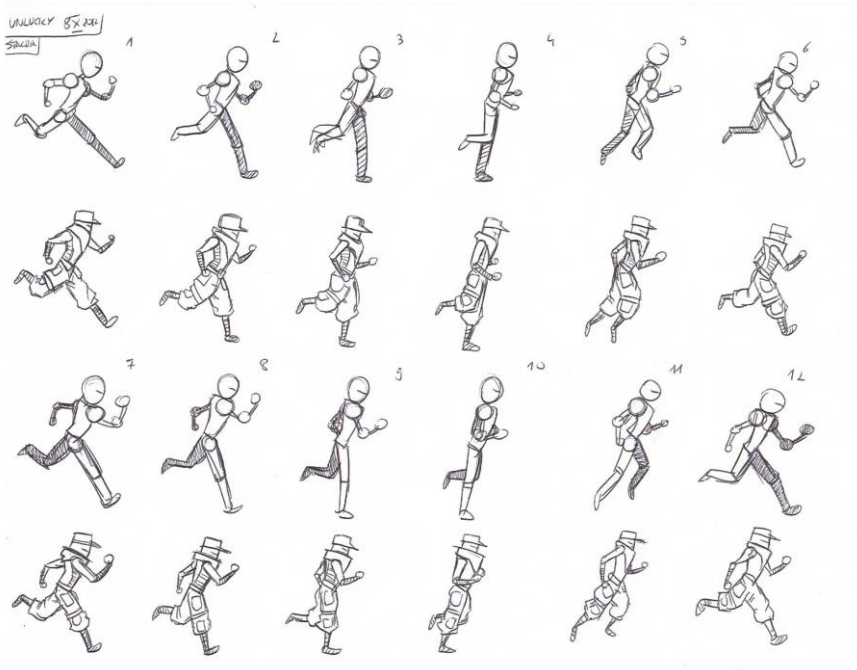
TWORZENIE SZATY GRAFICZNEJ APLIKACJI

Biorąc pod uwagę, że gra może zostać uruchomiona na urządzeniu, którego przekątna ekran może być mniejsza niż 4 cale, przyjęta stylistyka musiała być nie tylko czytelna ale i jednoznaczna.

Autor postanowił skupić się na samodzielnym wytwarzaniu potrzebnych grafik do gry. W pierwszej kolejności były tworzone szkice ogólne, warianty lokacji, przedmiotów (Rys. 4) i wyglądu postaci (Rys. 5).



Rys. 4: Concept art do gry Unlucky – projekty obiektów
(źródło: opracowanie własne)



Rys. 5: Concept art do gry Unlucky – projekty animacji
(źródło: opracowanie własne)

Następnie były tworzone wzorce, które po zeskanowaniu, mogły zostać wykorzystane do tworzenia odwzorowań i blueprintów[2] przy pomocy specjalistycznego oprogramowania graficznego Adobe PhotoShop. Kolejnym etapem było kolorowanie i komponowanie. Przygotowane w ten sposób grafiki były importowane do programu Construct 2, gdzie podlegały zamianie w sprite'y.

Tworzenie animacji wyglądało prawie tak samo, z tym że zamiast składać animację z kolejnych ujęć metodą animacji poklatkowej, dyplomant posłużył się programem do tworzenia animacji Toon Boom Studio na podstawie przesunięć wektorowych, które dopiero po właściwym ustawieniu i sparametryzowaniu nadawały się do wyeksportowania w formie animacji poklatkowej. W ten sposób można było wpływać na liczbę klatek animacji, przez ich dynamiczne wytwarzanie na podstawie elementów składowych modelu animowanego (części ciała) przedstawionego na Rys. 6, wektorów i modyfikatorów.

[2] Blueprint – rysunek techniczny, stanowiący graficzną formę dokumentacji struktury lub projektu. Pojęcie stosowane w branży gier wideo do określania rysunków bazowych, posiadających informacje na temat metryki, materiałów, proporcji obiektów itp.

Rys. 6: Blueprint do gry Unlucky – schemat budowy głównej postaci (źródło: opracowanie własne)

WYTWARZANIE SKRYPTOWYCH ELEMENTÓW SKŁADOWYCH I TWORZENIE EFEKTÓW WIZUALNYCH

Tworzenie skryptów WebGL wchodzi w skład ubogacania logiki aplikacji. Transformacje elementów graficznych, przy użyciu metod znanych z GLSL, dają okazję na zmianę postrzegania niektórych aspektów gry (Rys. 7 oraz 8).

Nie są to jednak zdarzenia ani funkcje niezbędne dla poprawnego działania aplikacji, ale za to bezpośrednio wpływają na ocenę jej estetyki i odpowiada za budowanie immersji.

Dodane do aplikacji efekty wizualne przy pomocy skryptów WebGL to:

- rozeta (ang. *rosette*) sprawia, że obraz zostaje przyciemniony gradientowo przy krawędziach;
- soczewka (ang. *lens*) sprawia, że widok jest zakrzywiony w taki sposób, że obiekty znajdujące się na środku ekranu są nieco większe, co ma na celu wzbudzenie poczucia, że użytkownik patrzy na ekran kineskopowy;
- poziome linie ekranu (ang. *scanlines*) sprawiają, że na ekranie pojawiają się poziome, powoli przesuwające się do dołu linie, odwzorowujące roztrojenie telewizora;
- dystorsje ekranu (ang.*distortions*) sprawiają, że na ekranie pojawiają się zakłócenia, odwzorowujące roztrojenie telewizora;

- rozłączanie kanałów RGBA (ang. *RGBA channel separation*) sprawia, że widok w grze jest dzielony na trzy kolory składowe: czerwony, zielony i niebieski;
- skala szarości (ang. *greyscale*) sprawia, że widok w grze może być sprowadzony do palety odcieni szarości;
- rozjaśnianie (ang. *brightness*) – widok w grze jest rozjaśniany;
- kontrast (ang. *contrast*) umożliwia zmianę kontrastu w widoku gry;
- szum (ang. *noise*) – w widoku gry pojawia się szum, charakterystyczny dla rozstrojonego telewizora lub źle ustawionej anteny;
- dyskretyzacja obrazu względem siatki o zmiennej wielkości (ang. pixellate) – sprawia, że obraz w grze może zostać zdyskretyzowany na przestrzeni kwadratów o bokach większych niż 1 piksel.

PROCES OPTYMALIZACJI DZIAŁANIA APLIKACJI

Ograniczone zasoby pamięci tymczasowej stanowią wąskie gardło dla płynnego i poprawnego działania aplikacji, a każda optymalizacja w kierunku lepszego zagospodarowania tą pamięcią jest na wagę złota, zwłaszcza w przypadku języków wysokiego poziomu, takich jak HTML czy JavaScript.

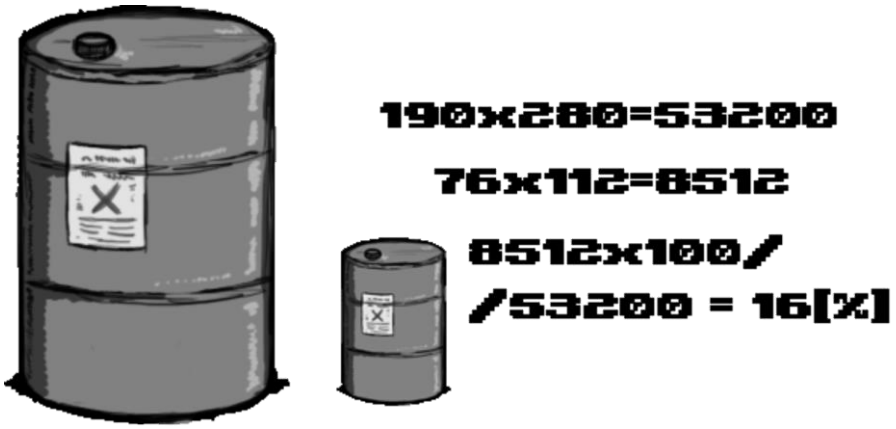
Optymalizacja elementów graficznych obejmowała wszystkie działania, które miały na celu obróbkę elementów graficznych aplikacji w taki sposób, aby zachowana została jakość wizualna tych elementów i aby wygląd całej gry nie uległ pogorszeniu. Ta część procesów optymalizacyjnych dotyczyła zagadnień opisanych poniżej.

W pierwszej kolejności należało zmniejszyć rozdzielczość obrazów. Wynikało to z faktu, że elementy umiejscowione na scenie z reguły są mniejsze od tych które zostały oryginalnie zaprojektowane. Redukcja rozmiaru nawet o 60% nie wpłynęła na zanik detali, ani spadek jakości elementów graficznych, natomiast doprowadziła do zmniejszenia zużycia pamięci potrzebnej na przetwarzanie tych elementów (lub inaczej – ilość danych potrzebnych do zapamiętania obrazka) o około 80%.

Rys. 7. Widok aplikacji w przeglądarce nie wspierającej WebGL (źródło: opracowanie własne)

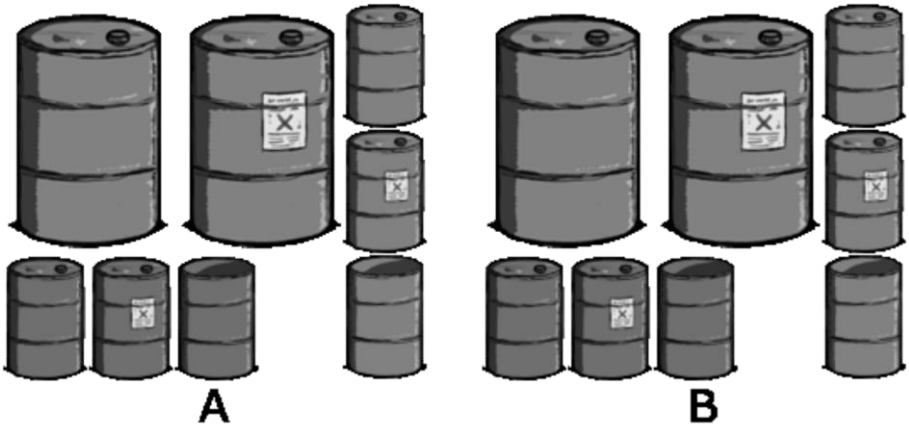
Rys. 8. Widok aplikacji w przeglądarce wspierającej WebGL (źródło: opracowanie własne)

Na Rys 9. przedstawione zostało zestawienie grafiki z projektu z grafiką zmniejszoną na potrzeby aplikacji. Oryginalny projekt zawiera 53200 pikseli, zaś zmniejszony już tylko 8512. Oznacza to, że dla tego konkretnego obrazka udało się osiągnąć optymalizację zużycia pamięci potrzebnej na jego przetworzenie o 84%.



Rys. 9: Przykładowe zestawienie grafiki przed zmianą rozdzielczości i po (źródło: opracowanie własne)

Na Rys. 10 przedstawione zostało zestawienie przykładowej grafiki przed i po procesie jej optymalizacji. Walory wizualne nie uległy zmianie, natomiast rozmiar zajmowany na dysku i w pamięci podręcznej już tak.
Obrazek przedstawiony powyżej, przed optymalizacją zajmował 162KB, po optymalizacji już tylko 36KB. Oznacza to wzrost efektywności pracy na tym jednym elemencie o ponad 87%.



Rys. 10: Przykładowe zestawienie grafiki przez optymalizacją (A) i po optymalizacji (B) (źródło: opracowanie własne)

Na podstawie wyników z przeprowadzenia procesów optymalizacji, oraz na podstawie własnych doświadczeń nabytych w czasie przyprowadzania tych procesów, dyplomant doszedł do wniosku, że mając do czynienia z technologią wysokiego poziomu, nie można nigdy zapominać o procesach i metodach optymalizacji. Każda nawet najmniejsza optymalizacja w tym przypadku ma ogromne znaczenie.

Wszystkie podejmowane działania muszą być zawsze dokładnie przemyślne gdyż każda nawet najmniej istotna operacja, może się przełożyć na późniejsze gorsze działanie wszystkich pozostałych elementów składowych aplikacji.

TESTY WYDAJNOŚCIOWE

W czasie tworzenia aplikacji z wykorzystaniem silnika Construct 2, kluczowe okazały się regularne testy wydajnościowe aplikacji, służące sprawdzeniu efektów wprowadzanych modyfikacji. Zazwyczaj, wprowadzane modyfikacje dawały pozytywne rezultaty w postaci zwiększenia ilości klatek animacji odtwarzanych w ciągu sekundy.

Test wydajnościowy, sprawdzający prędkość działania i stabilność aplikacji, polegał na uruchomieniu jej na różnych urządzeniach, różniących się między sobą specyfikacją sprzętową i zainstalowanym systemem operacyjnym.

Wyniki testów nie zawsze wskazywały na widoczny postęp w czasie dokonywania działań optymalizacyjnych. Prawdziwy postęp można za to zaobserwować przy porównywaniu wyników sprzed rozpoczęcia cyklu działań optymalizacyjnych z najnowszymi. Takie zestawienie daje łatwe i jednoznaczne dane do interpretacji.

Tabela 1 zawiera dane otrzymane z obserwacji średniej ilości klatek na sekundę w czasie pracy aplikacji, przy minimalnym obciążeniu urządzeń innymi procesami. Udało się stwierdzić średni wzrost wydajności, który dzięki procesom optymalizacyjnym wynosi ok. 250%. Testy wydajnościowe były przeprowadzane na urządzeniach wyposażonych w najnowsze dostępne wersje systemów operacyjnych.

Tab. 1. Zestawienie testów wydajnościowych sprawdzających średnią ilość klatek na sekundę (opracowanie własne)

	Średnia liczba klatek na sekundę (FPS)	
model telefonu	**Przed optymalizacją**	**Po optymalizacji**
Google Nexus 4	3	10
LG Swift L5	4	10
Apple Iphone 5	5	15
Samsung Galaxy S	5	12
Samsung Galaxy S2	4	16
Nokia Lumia 710	2	11
Asus Transformer	5	15

WNIOSKI

Przyjęty cel został osiągnięty. Zostało jasno udowodnienie, że wytworzenie aplikacji spełniającej wspomniane wymogi jest możliwe i że taka aplikacja może zostać uruchomiona na urządzeniach mobilnych. To czego nie udało się uzyskać jest wyższy stopień zaawansowania technicznego projektu. Wynika to z niedostatecznego zorganizowania pracy nad projektem, braku części narzędzi, umiejętności, oraz czasu potrzebnego na testy i prowadzenie procesów optymalizacyjnych, połączonych z nauką nowych technik programistycznych.

Oprogramowanie stworzone przez firmę Scirra dobrze spełnia się w tej roli, a sami użytkownicy nie narzekają na kierunek rozwoju firmy i samego programu. Można stąd wywnioskować, że droga jaką obrał zespół programistów przy początku tworzenia programu Construct 2 jak dotąd sprawdza się.

Obecnie jest to jedyny produkt na rynku, który posiada tak duży zbiór narzędzi potrzebnych do tworzenia gier w HTML5, a jednocześnie umożliwia samodzielne rozbudowywanie środowiska o nowe funkcjonalności.

INFORMACJE O AUTORACH

KAROL BANASZKIEWICZ	student kierunku Informatyka Wydział Elektrotechniki i Informatyki Politechnika Lubelska
PAWEŁ HAWRYLAK	student kierunku Informatyka Wydział Elektrotechniki i Informatyki Politechnika Lubelska
KAMIL KOŁODZIEJCZYK	student kierunku Informatyka Wydział Elektrotechniki i Informatyki Politechnika Lubelska
EDYTA ŁUKASIK	Instytut Informatyki Wydział Elektrotechniki i Informatyki Politechnika Lubelska
MATEUSZ MICHALCZYK	student kierunku Informatyka Wydział Elektrotechniki i Informatyki Politechnika Lubelska Transition Technologies S.A.
MACIEJ MIŁOSZ	student kierunku Informatyka Wydział Elektrotechniki i Informatyki Politechnika Lubelska Transition Technologies S.A.
KAMIL NOWAK	student kierunku Informatyka Wydział Elektrotechniki i Informatyki Politechnika Lubelska
BEATA PAŃCZYK	Instytut Informatyki Wydział Elektrotechniki i Informatyki Politechnika Lubelska

MACIEJ PAŃCZYK	Instytut Informatyki Wydział Elektrotechniki i Informatyki Politechnika Lubelska
MAŁGORZATA PLECHAWSKA-WÓJCIK	Instytut Informatyki Wydział Elektrotechniki i Informatyki, Politechnika Lubelska
STANISŁAW SKULIMOWSKI	Instytut Informatyki Wydział Elektrotechniki i Informatyki Politechnika Lubelska
ALEKSANDER WOJDYGA	Transition Technologies S.A.

INFORMACJA O KOŁACH NAUKOWYCH UCZESTNICZĄCYCH W IV SYMPOZJUM NAUKOWYM ELEKTRYKÓW I INFORMATYKÓW

KOŁO NAUKOWE AUTOMATYKI

Koło Naukowe Automatyki jest kołem działającym przy Katedrze Automatyki i Metrologii na Wydziale Elektrotechniki i Informatyki, Politechniki Lubelskiej. Aktualnym opiekunem koła jest dr inż. Adam Kurnicki.

Głównymi zagadnieniami, jakimi zajmują się członkowie koła są badania związane z syntezą algorytmów sterowania szeroko pojętymi układami robotycznymi. Innym chętnie podejmowanym zagadnieniem, jest implementowanie algorytmów sterowania procesami przemysłowymi na sterownikach PLC.

Obecnie do Koła Naukowego Automatyki należy kilkanaścioro studentów politechniki. Spotkania członków koła oraz badania odbywają się w Laboratorium Zaawansowanych Układów Sterowania. Laboratorium posiada aparaturę kontrolno-pomiarową umożliwiającą projektowanie i prototypowanie w czasie rzeczywistym układów sterowania zaawansowanymi układami mechatronicznymi.

Studenci koła rokrocznie uczestniczą w konferencjach naukowych prezentując swoje prace dotyczące zagadnień związanych z automatyką i robotyką.

Więcej informacji i aktualności dotyczących działalności koła KNA można znaleźć na stronie http://elektron.pol.lublin.pl/users/kna/index.htm.

Studenckie Koło Naukowe Elektroekologów ELMECOL jest kołem działającym przy Instytucie Podstaw Elektrotechniki i Elektrotechnologii (IPEiE) Wydziału Elektrotechniki i Informatyki, Politechniki Lubelskiej. Aktualnym opiekunem koła jest dr inż. Paweł Mazurek.

Głównymi zagadnieniami, jakimi zajmują się członkowie koła są badania kompatybilności elektromagnetycznej urządzeń elektrycznych, pomiary i analiza natężeń pól elektromagnetycznych niskiej i wysokiej częstotliwości oraz badania emisji hałasu.

Obecnie do Koła Naukowego ELMECOL należy kilkanaścioro studentów Politechniki. Spotkania członków koła oraz badania odbywają się w Laboratorium Kompatybilności Elektromagnetycznej istniejącym przy Instytucie IPEiE w budynku ASPPECT. Laboratorium posiada aparaturę pomiarową umożliwiającą przeprowadzenie pomiarów ekspozycji pól elektromagnetycznych niskiej i wysokiej częstotliwości.

Studenci koła aktywnie uczestniczą w konferencjach naukowych dotyczących zagadnieniom emisji elektromagnetycznej. Dzięki dofinansowaniu prorektora Politechniki Lubelskiej, studenci koła uczestniczą już od kilku lat w międzynarodowym sympozjum naukowym Polskiego Towarzystwa Zastosowań Elektromagnetyzmu oraz na konferencji Forum Inżynierii Ekologicznej, gdzie prezentowali już wiele artykułów z tematyki pomiarów natężeń pól elektrycznych i magnetycznych.

Więcej informacji i aktualności dotyczących działalności koła ELMECOL można znaleźć na stronie http://elmecol.pollub.pl.

Koło naukowe Elektroników i Mechatroników SEMICON jest kołem działającym przy Instytucie Elektroniki i Technik Informacyjnych Wydziału Elektrotechniki i Informatyki, Politechniki Lubelskiej. Opiekunem koła naukowego jest dr inż. Andrzej Kociubiński przy współpracy z dr inż. Mariuszem Dukiem.

Działalność koła naukowego zmierza do rozwoju zainteresowań i zdobywania nowych umiejętności przez studentów z kierunków Elektrotechnika, Informatyka i Mechatronika zainteresowanych projektowaniem systemów elektronicznych, technologią cyfrową oraz robotyką. Naukowe cele koła obejmują zastosowanie nowoczesnych technologii mikroprocesorowych w projektowaniu systemów elektronicznych, budowy autonomicznych urządzeń elektronicznych oraz wykorzystaniu najnowszych technologii półprzewodnikowych. Dzięki współpracy z zewnętrznymi firmami, studenci mają możliwość zdobycia doświadczenia oraz nawiązania kontaktów zawodowych owocujących w przyszłości.

W chwili obecnej w pracach koła naukowego SEMICON uczestniczy kilkunastu studentów politechniki. Spotkania koła zwykle odbywają się na Wydziale Elektrotechniki i Informatyki Politechniki Lubelskiej.

Członkowie koła naukowego SEMICON biorą aktywny udział w konferencjach poświęconych tematyce związanej z pracami koła. Uczestniczyliśmy w Sympozjum Naukowym Elektrotechników i Informatyków organizowanym przez Politechnikę Lubelską oraz IEEE-SPIE Joint Symposium Wilga 2013 organizowanego przez Politechnikę Warszawską. Udział w sympozjach umożliwia studentom rozwój w interesujących ich dziedzinach oraz integrację ze środowiskiem naukowym.

Dodatkowe informacje oraz aktualności dotyczące prac koła można znaleźć na stronie internetowej: http://semicon.pollub.pl/.

Koło Naukowe Elektroników MICROCHIP działa przy instytucie Elektroniki i Technik Informacyjnych Politechniki Lubelskiej. W poprzednich latach opiekunami Koła byli dr inż. Wojciech Surtel oraz mgr inż. Krzysztof Król. Obecnie funkcję tą pełni mgr inż. Marcin Maciejewski.

Celem działalności Koła jest kształcenie z zakresu projektowania układów elektronicznych oraz programowania mikrokontrolerów. Realizowane jest to poprzez wewnętrzne szkolenia, organizowane przez członków Koła oraz uczestniczenie w projektach, związanych z budową robotów mobilnych. W ostatnim czasie Koło organizowało szkolenia z zakresu programowania mikrokontrolerów AVR w języku C. Dodatkowo prowadziło szkolenia z podstaw wiedzy elektronicznej, a także projektowania układów elektronicznych analogowych i cyfrowych. Obecnie skupia się na projekcie budowy robota mobilnego na zawody ERC 2014.

Zebrania Koła Naukowego MICROCHIP odbywają się we środy o godzinie 18 w Sali S10 w Instytucie Informatyki Politechniki Lubelskiej. Koło zrzesza 16 osób, wśród których znajdują się studenci Elektrotechniki, Informatyki i Mechatroniki (Wydział Elektrotechniki i Informatyki), a także Inżynierii Biomedycznej oraz Mechaniki i Budowy Maszyn (Wydział Mechaniczny Politechniki Lubelskiej).

Więcej informacji dotyczących KN MICROCHIP można znaleźć na stronie: http://microchip.politechnika.lublin.pl/

Koło Naukowe Elektrycznych Systemów Naukowych „ZORDON" jest kołem działającym przy Katedrze Inżynierii Komputerowej i Elektrycznej na Wydziale Elektrotechniki i Informatyki Politechniki Lubelskiej.
Jednym z założycieli i aktualnym opiekunem koła jest dr inż. Marek Horyński.

Głównym celem działalności Koła jest zdobywanie i rozpowszechnianie wśród studentów wiedzy dotyczącej współczesnego rynku automatyki budynkowej oraz przemysłowej.

Aktywność koła koncentruje się na następujących działaniach:

- prace badawcze związane z działaniem i strukturą systemów inteligentnych automatyki budynkowej,
- prace badawcze związane z integracją instalacji inteligentnych z innymi systemami automatyki w budynkach,
- organizacja i udział w szkoleniach z zakresu Elektrycznych Systemów Inteligentnych.

W szkoleniach tych biorą udział Partnerzy koła z przemysłu: Hager, LCN, ABB, MCD Electronics.

Obecnie do Koła Naukowego „Zordon" należy około dziesięciu studentów Politechniki. Spotkania członków koła oraz badania odbywają się w Laboratorium Elektrycznych Systemów Inteligentnych istniejącym przy Katedrze Inżynierii Komputerowej i Elektrycznej, w budynku przy ul. Okopowej 8. Laboratorium posiada aparaturę pomiarową oraz modele systemów inteligentnych budynków umożliwiające przeprowadzenie badań i symulacji różnych układów automatyki.

Studenci koła aktywnie uczestniczą w konferencjach naukowych i szkoleniach dotyczących nowoczesnego budownictwa.

Więcej informacji i aktualności dotyczących działalności koła „ZORDON" można znaleźć na stronie http://zordon.pollub.pl.

Koło Naukowe Informatyki „Grupa .Net Politechniki Lubelskiej" jest Kołem działającym przy Instytucie Informatyki na Wydziale Elektrotechniki i Informatyki Politechniki Lubelskiej. Aktualnym opiekunem koła jest mgr inż. Marcin Badurowicz.

Głównymi zagadnieniami, jakimi zajmują się członkowie Koła jest nauka tworzenia aplikacji dla platform Windows, Windows Phone oraz Windows Azure z wykorzystaniem języka C#. Prowadzone są prezentacje na temat tych dziedzin jak również inżynierii oprogramowania, testów, programowania gier i wielu innych.

Obecnie do Grupy .Net należy kilkanaścioro studentów Politechniki, niemniej cotygodniowe spotkania w formie wykładów oraz warsztatów gromadzą szeroką publiczność osób nie zawsze związanych z rdzeniem Koła. Prowadzący wykłady i warsztaty to zarówno studenci, jak i zaproszeni pracownicy zaprzyjaźnionych firm. Spotkania najczęściej odbywają się w jednej z sal wykładowych budynku Instytutu Informatyki (Pentagon).

Studenci Koła organizują między innymi studencką konferencję informatyczną IT Academic Day, której zeszłoroczna edycja zgromadziła ponad setkę studentów z lubelskich uczelni; organizowane są również całodniowe warsztaty CodeCamp, w ramach których budowane są aplikacje i gry dostępne później dla platform Windows.

Więcej informacji i aktualności dotyczących działalności Grupy .Net można znaleźć na stronie http://codeguru.geekclub.pl/grupy/developers.

Koło Naukowe Informatyki PENTAGON działa przy Instytucie Informatyki na Wydziale Elektrotechniki i Informatyki Politechniki Lubelskiej.

W latach 1999 – 2010 opiekunem Koła był dr inż. Marek Miłosz, zaś od roku 2011 rolę tą pełni mgr inż. Maciej Laskowski.

Obecnie Koło Naukowe Informatyki PENTAGON zajmuje się głównie grafiką i modelowaniem 3D oraz różnego rodzaju technikami multimedialnymi, nie stroniąc jednak od rozwiązań programistycznych..

Koło wspólnie z Urzędem Miasta organizowało również darmowe szkolenia z Blendera – darmowego programu do tworzenia grafiki 3D.

Tradycyjnie od wielu lat spotkania Koła odbywają się w każdy czwartek o godzinie 20.00 w sali S111 w Instytucie Informatyki Politechniki Lubelskiej.

Członkowie Koła uczestniczą w wielu konferencjach krajowych i międzynarodowych (Inżynieria Gier Komputerowych, Gameday, Konferencja Matematyki i Informatyki Stosowanej, Studencki Festiwal Informatyczny) oraz od samego początku współorganizują Sympozjum Naukowe Elektryków i Informatyków, zapewniając m.in. obsługę oficjalnej strony Sypozjum.

Na chwilę obecną Koło zrzesza studentów zarówno z Wydziału Elektrotechniki i Informatyki Politechniki Lubelskiej, jak również studentów z innych uczelni, m.in. Uniwersytetu Marii Curie-Skłodowskiej czy Katolickiego Uniwersytetu Lubelskiego.

Więcej informacji i aktualności dotyczących działalności koła KNIP można znaleźć na stronie http://knip.pol.lublin.pl.

Studenckie Koło Naukowe Materiałoznawstwa Elektrycznego i Techniki Wysokich Napięć MELJON działa przy Katedrze Urządzeń Elektrycznych i Techniki Wysokich Napięć na Wydziale Elektrotechniki i Informatyki. Opiekunem koła jest dr inż. Tomasz N. Kołtunowicz. Obecnie do Koła Naukowego MELJON należy 11 studentów z Wydziału Elektrotechniki i Informatyki Politechniki Lubelskiej.

Głównymi zagadnieniami, jakimi zajmują się członkowie Koła to prace nad:

- badaniami właściwości elektrycznych nanokompozytów o strukturze metal-dielektryk, w których to cząsteczki fazy metalicznej rozmieszczone są losowo w matrycy izolacyjnej z tlenków lub fluorków metali (głównie Al_2O_3, CaF_2 oraz PZT);
- badaniami dotyczącymi nieniszczących metod kontroli stanu izolacji wysokonapięciowych transformatorów energetycznych;
- budową stanowiska pomiarowego do badania izolacji papierowo-olejowej metodą PDC;
- modyfikacją stanowiska z implantatorem jonów;
- budową oraz uruchomieniem stanowiska pokazowo-diagnostycznego wyposażonego w transformator Tesli.

Członkowie koła aktywnie uczestniczą w konferencjach i sympozjach naukowych, m. in. w międzynarodowych konferencjach ION oraz NEET, Ogólnopolskiej Konferencji Studenckiej „Nowoczesne Metody Doświadczalne Fizyki, Chemii i Inżynierii", a także czynnie uczestniczą oraz współorganizują Sympozjum Naukowe Elektryków i Informatyków SNEiI.

Więcej informacji i aktualności dotyczących działalności koła MELJON można znaleźć na stronie http://meljon.pollub.pl.

Koło Naukowe MECHATRONIK jest kołem działającym przy Katedrze Napędów i Maszyn Elektrycznych na Wydziale Elektrotechniki i Informatyki Politechniki Lubelskiej. Aktualnym opiekunem Koła jest mgr inż. Krzysztof Jahołkowski.

Głównymi zagadnieniami, jakimi zajmują się członkowie Koła są projekty dotyczące zastosowania sterowników PLC do sterowania układami napędowymi prądu stałego i przemiennego, zastosowania programów do wizualizacji i zbierania danych pomiarowych oraz zastosowania ogniw fotowoltaicznych do zasilania generatorów plazmy niskotemperaturowej.

Obecnie do Koła Naukowego MECHATRONIK należy kilkanaścioro studentów Politechniki Lubelskiej. Spotkania członków Koła oraz badania odbywają się w Laboratorium Maszyn Elektrycznych oraz Laboratorium Modelowania i Symulacji Komputerowej istniejących przy Katedrze Napędów i Maszyn Elektrycznych. Laboratoria posiadają aparaturę pomiarową umożliwiającą przeprowadzenie badań i pomiarów układów napędowych oraz układów zasilania generatorów plazmy niskotemperaturowej.

Studenci Koła aktywnie uczestniczą w sympozjach SNEiI, gdzie prezentowali artykuły z tematyki realizowanych projektów.

Więcej informacji i aktualności dotyczących działalności Koła MECHATRONIK można znaleźć na stronie http//proton.pol.lublin.pl/mt/.

Koło Naukowe Elektryków "Napęd i Automatyka" działa przy Katedrze Napędów i Maszyn Elektrycznych na Wydziale Elektrotechniki i Informatyki Politechniki Lubelskiej. Opiekunem koła jest dr inż. Piotr Filipek.

Koło skupia studentów Wydziału Elektrotechniki i Informatyki różnych kierunków i specjalności, pragnących poszerzać swoją wiedzę poprzez udział w sympozjach naukowych, seminariach naukowo-technicznych, warsztatach laboratoryjnych i wycieczkach do przedsiębiorstw i instytutów naukowych.

Podstawową formą pracy w Kole są warsztaty laboratoryjne, w których studenci mają możliwość rozwijania swoich zainteresowań. W laboratoriach Katedry Napędów Elektrycznych poznają nowoczesne urządzenia zautomatyzowanych napędów elektrycznych, badają i opracowują własne projekty.

Zainteresowania członków Koła podzielić można na pięć obszarów tematycznych:

- Układy napędowe zasilane z fotowoltaicznych źródeł energii
- Układy mikroprocesorowego sterowania napędami elektrycznymi, w tym układy z DSP
- Sterowanie dyskretne systemów napędowych w zautomatyzowanych układach sterowania PLC
- Problemy optymalnego wyboru układu napędowego dla zespołu technologicznego
- Elektrownie wiatrowe – wybrane zagadnienia z zakresu sterowania pracą generatorów i analizy ekonomicznej inwestycji.

Więcej informacji i aktualności dotyczących działalności koła KNNA można znaleźć na stronie http://knna.pollub.pl oraz http://ELVIC.pollub.pl.

Studenckie Koło Naukowe Pentagon Café jest kołem działającym przy Instytucie Informatyki Wydziału Elektrotechniki i Informatyki, Politechniki Lubelskiej. Aktualnym opiekunem koła jest dr inż. Piotr Kopniak.

Członkowie koła zajmują się badaniami technologii wytwarzania oprogramowania. W ramach zajęć Koła rozwijają internetowy system wspomagania pracy dydaktycznej jednostki uczelni wyższej, który wykorzystywany jest w Instytucie Informatyki. Rozwój systemu wymaga badań związanych z analizą wymagań nowej funkcjonalności, tworzeniem projektów, zarządzaniem kodem źródłowym, wersjonowaniem i dystrybucją, a także pracą zespołową i zarządzaniem całym projektem.

Obecnie do Koła Naukowego Pentagon Café należy ośmioro studentów Politechniki Lubelskiej. Spotkania członków koła oraz badania odbywają się w salach wykładowych i laboratoriach komputerowych przy Instytucie Informatyki.

Wyniki prac Koła prezentowane były w postaci artykułów naukowych i prezentacji na konferencjach naukowych i dydaktycznych.

Więcej informacji dotyczących działalności Koła Pentagon Café można znaleźć na stronie http://cafe.pollub.pl.

Studenckie Koło Naukowe Zastosowań Technologii .NET jest kołem działającym przy Instytucie Elektroniki i Technik Informacyjnych Wydziału Elektrotechniki i Informatyki, Politechniki Lubelskiej. Aktualnym opiekunem koła jest mgr inż. Daniel Sawicki.

Głównymi zagadnieniami, jakimi zajmują się członkowie koła jest przyśpieszenie wykonywania obliczeń i przetwarzania danych z wykorzystywaniem najnowszych technologii programistycznych. Koło Naukowe zajmuje się klastrami i chmurami obliczeniowymi oraz bardzo wydajnym przetwarzanie za pomocą kart graficznych.

Obecnie do Koła Naukowego Zastosowań Technologii .NET należy kilkanaścioro studentów politechniki. Spotkania członków koła oraz badania odbywają się w Laboratorium Sieci Komputerowych istniejącym przy Instytucie Elektroniki i Technik Informacyjnych w budynku PENTAGON. Laboratorium posiada sprzęt komputerowy wyposażony w wielordzeniowe procesory oraz karty graficzne wykorzystujące technologię CUDA. Koło Naukowe jest w posiadaniu wydajnej karty do obliczeń Tesla C1030.

Więcej informacji i aktualności dotyczących działalności Koła Naukowego Zastosowań Technologii .NET można znaleźć na stronie http://dotnet.politechnika.lublin.pl.

Studenckie Koło Naukowe KERNEL.C jest kołem działającym przy Zakładzie Informatyki Wydziału Transportu i Informatyki, Wyższej Szkoły Ekonomii i Innowacji w Lublinie. Aktualnym prezesem koła jest Mirosław Smoczyński

Głównym zagadnieniem, jakimi zajmują się członkowie koła jest przetwarzanie danych multimedialnych w środowiskach rozproszonych i wirtualnych. Koło naukowe zajmuje się przygotowaniem danych multimedialnych (odpowiednie kodowanie i kompresja) i badaniem wydajności przetwarzania tych danych w czasie transmisji przez sieć komputerową z wykorzystaniem różnych środowisk.

Obecnie do Koła Naukowego KERNEL.C należy kilku studentów Wyższej Szkoły Ekonomii i Innowacji. Spotkania członków koła oraz badania odbywają się w Laboratorium bezpieczeństwa usług sieciowych istniejącym na WSEI. Laboratorium posiada serwer modułowy Modular Server SH-8614 V7, który jest bardzo wydajną jednostką obliczeniową.

Więcej informacji i aktualności dotyczących działalności koła KERNEL.C będzie można znaleźć na powstającej stronie http://kernelc.wsei.lublin.pl.

INFORMACJE O IV SYMPOZJUM NAUKOWYM ELEKTRYKÓW I INFORMATYKÓW

ORGANIZATORZY IV SYMPOZJUM NAUKOWEGO ELEKTRYKÓW I INFORMATYKÓW

1. Urząd Miasta Lublin
2. Koło Naukowe Elektryków „NAPĘD i AUTOMATYKA”
3. Koło Naukowe Elektroekologów „ELMECOL”
4. Koło Naukowe Informatyki „PENTAGON”
5. Koło Naukowe Materiałoznawstwa Elektrycznego i Techniki Wysokich Napięć „MELJON”
6. Studencka Sekcja Stowarzyszenia Elektryków Polskich
7. Samorząd Studencki Politechniki Lubelskiej

KOMITET NAUKOWY IV SYMPOZJUM NAUKOWEGO ELEKTRYKÓW I INFORMATYKÓW

1. prof. dr hab. inż. Henryka D. Stryczewska – przewodnicząca
2. prof. dr hab. inż. Piotr Kacejko – JM Rektor Politechniki Lubelskiej
3. prof. dr hab. inż. Jan Sikora
4. prof. dr hab. inż. Antoni Świć
5. prof. dr hab. inż. Andrzej Wac–Włodarczyk
6. prof. dr hab. inż. Waldemar Wójcik
7. dr hab. Stanisław Grzegórski, prof. PL
8. dr hab. inż. Wojciech Jarzyna, prof. PL
9. dr hab. inż. Czesław Karwat, prof. PL
10. dr hab. inż. Piotr Kisała, prof. PL
11. dr hab. inż. Jan Kolano, prof. PL
12. dr hab. inż. Andrzej Kotyra, prof. PL
13. dr hab. inż. Jerzy Montusiewicz, prof. PL
14. dr hab. inż. Jarosław Sikora, prof. PL
15. dr hab. inż. Paweł Surdacki, prof. PL
16. dr inż. Marek Miłosz

KOMITET ORGANIZACYJNY IV SYMPOZJUM NAUKOWEGO ELEKTRYKÓW I INFORMATYKÓW

1. dr inż. Piotr Z. Filipek – przewodniczący
2. dr inż. Paweł A. Mazurek
3. mgr inż. Maciej Laskowski
4. dr inż. Tomasz N. Kołtunowicz

SPONSORZY IV SYMPOZJUM NAUKOWEGO ELEKTRYKÓW I INFORMATYKÓW

DZIĘKUJEMY!!!

PATRONI IV SYMPOZJUM NAUKOWEGO ELEKTRYKÓW I INFORMATYKÓW

Patronat Prezydenta Miasta Lublin

PATRONAT HONOROWY | PREZYDENT MIASTA LUBLIN KRZYSZTOF ŻUK

Patronat honorowy Prezesa Urzędu Komunikacji Elektronicznej

Patronat Lubelskiego Oddziału Stowarzyszenia Elektryków Polskich

Patronat Jego Magnificencji Rektora Politechniki Lubelskiej

Patronat Dziekana Wydziału Elektrotechniki i Informatyki

Patronat Polskiego Towarzystwa Informatycznego – Koło w Lublinie

Patronat Lubelskiego Oddziału Polskiego Towarzystwa Zarządzania Produkcją