Jan K. Argasiński

Software Studies.

Kultura, sztuka i oprogramowanie.

Praca doktorska napisana pod kierunkiem prof. dr. hab. Andrzeja Pitrusa.

Uniwersytet Jagielloński w Krakowie.
Wydział Zarządzania i Komunikacji Społecznej.

Kraków 2014

Wstęp: Programuj albo zostaniesz zaprogramowany. 1

Część 1: System (Σύστημα). 4

Σύστημα 1: Wyznaczanie paradygmatu.
Epistemologia jako teoria informacji. 4

Σ 1.1 Epistemologia. 5

Σ 1.2 Prawda; Prawdy, Prawa. 6

Σ 1.3 Logika i matematyka. W poszukiwaniu fundamentu. 7

Σ 1.4 Maszyna Turinga. Logika w działaniu. 9

Σ 1.5 Epistemologia jako teoria informacji. 12

Σύστημα 2: Informacja, dane, wiedza.
Kod i struktura. 15

Σ 2.1 Problem pojęcia „informacji”. 16

Σ 2.2. Ilościowa miara informacji w teorii komunikacji. Kluczowe pojęcia. 18

Σ 2.2.1 Usystematyzowanie pojęć: miary informacji. 22

Σ 2.3 Dane a semantyczna koncepcja informacji. 28

Σ 2.4 W stronę komputacji. Epistemologia jako teoria informacji 2. 32

Σ 2.Apendyks: Jakiego kodu użyć? 35

Σύστημα 3: Komputer i język.
Od alfabetu do algorytmu. 40

Σ 3.1 Źródła. W Poszukiwaniu języka uniwersalnego. 41

Σ 3.1.1 Język filozoficzny Wilkinsa i lingua characteristica Leibniza. 42

Σ 3.2 Język i algorytm. 47

Interfejs między częściami 1:
Od Σ do A: Maszyny semiotyczne. 51

Od Σ do A 1: Pytanie o „semiotykę komputerową”. 51

Od Σ do A 2: Maszyna syntaktyczna. Język jako model obliczeniowy. 53

Od Σ do A 3: Dwa wektory. 58

Część 2: Analityka (Αναλυτικά). 59

Αναλυτικά 1: Język maszynowy. 59

A 1.1 Procesor. 60

A 1.2 Instrukcja po instrukcji. Bit po bicie. 62

Αναλυτικά 2:
Struktury danych + algorytmy = programy (software). 67

A 2.1 Struktury danych. 68

A 2.2. Algorytmy. 78

A 2.2.1 Schematy blokowe algorytmów. Procedury i rekurencja. 80

A 2.2.2 Sztuka algorytmiczna / sztuka generatywna. 83

A 2.2.3 Algorytmy w digital humanities. 89

A 2.2.4 Znaczenie algorytmu. 95

A 2.3 Programy / oprogramowanie (software). 101

A 2.3.1 Oprogramowanie jako sztuka. Kilka słów o demoscenie. 102

Αναλυτικά 3:
Od Assemblera przez Lisp i Prolog po Wolfram Language i Swift.
Poziomy języków programowania. 119

A 3.1 Prawdziwi Programiści 120

A 3.2 BASIC(s) czyli podstawy. 122

A 3.3 Hakerzy i malarze. 124

A 3.4 Uniwersum programowania (bardzo)wysokopoziomowego. 132

Αναλυτικά 4: Paradygmaty programowania.
Od kodu maszynowego do obiektowych ontologii. 137

A 4.1. Programowanie imperatywne. 138

A 4.2 Programowanie deklaratywne. 140

Αναλυτικά 5: Critical Code Studies. 147

Αναλυτικά 6: Dane, wyszukiwanie.
Potop informacyjny, Sieć Semantyczna, Calculus Ratiocinator
i Wolfram Alpha. 159

Αναλυτικά 7: Pamięć operacyjna. Pamięć kulturowa. 168

Αναλυτικά 8: Przyszłość teraz.
Ubiquitous Computing & Internet of Things. 174

Interfejs między częściami 2:
Od A do Δ: There is no software. 184

Część 3: Dynamiki (Δυναμική) 187

Δυναμική 1: Oprogramowanie jako kultura. 187

Δυναμική 2: Paradygmat rozproszony.
Sieć, protokoły komunikacyjne. 190

Δυναμική 2A: Transdukcja.
Dynamika kodu i przestrzeni. 200

Δυναμική 3: Partycypacyjność.
Od hakera do użytkownika i z powrotem. 205

Interfejs między częściami 3:
Od Δ do T: Digital humanities.
Oprogramowanie kultury i hermeneutyka informatyki. 207

Część 4: Techne (Τέχνη). 208

Τέχνη 1: Humanistyka cyfrowa i zwrot komputacyjny. 208

Τέχνη 2: Media Lab Toolkit.
Narzędzia, problemy, przypadki użycia. 213

Zakończenie: Zajrzeć pod powierzchnię ekranu. 230

Załączniki: kody źródłowe etc. 232

Załącznik 1:

Gra w życie (kod źródłowy Processing) 232

Załącznik 2:

Sortowanie Bąbelkowe (kod źródłowy Processing) 236

Załącznik 3:

Sea and Spar Beetween (kod źródłowy Java Script) 237

Załącznik 4: 249

Hello World (kod źródłowy Shakespeare) 249

Bibliografia: 253

Wstęp: Programuj albo zostaniesz zaprogramowany.

Dzisiejszy świat jest zdominowany przez oprogramowanie. Jego działanie wywiera wpływ na niemal wszystkie dziedziny naszego zarówno indywidualnego jak i zbiorowego życia codziennego - w tym na sferę kreatywną i twórczość artystyczną. Z perspektywy nawet wyspecjalizowanego użytkownika komputer jawi się przede wszystkim jako narzędzie, zaś oprogramowanie jako jedna z jego części - ta, która umożliwia wykonanie konkretnego, pożądanego działania. Już takie postawienie sprawy powoduje, że kwestia konstruowania i działania *softwaru* staje się bezpośrednim przedmiotem zainteresowania kulturoznawstwa, medioznawstwa i nauk o sztuce. Pouczają nas wszak te dziedziny, że nie ma czegoś takiego jak *niewinne* używanie narzędzi. Ich struktura determinuje sposób realizowania przez nas aktywności, z którymi są związane, wpływając trwale zarówno na przebieg wykonywanej czynności jak i finalny efekt. Wybrane narzędzia stają się też dominującymi metaforami, za pomocą których wyjaśnia się rzeczywistość. Jak zwraca uwagę J. David Bolter każda epoka w dziejach ludzkości ma swoją *technologię definiującą,* która jest nie tylko bardzo ważna z perspektywy praktycznego zastosowania ale przede wszystkim zmienia ona sposób postrzegania świata. Starożytność miała koło garncarskie i inne technologie manualne, nowożytność zachwyciła się najpierw techniczną elegancją pracy zegara, potęgą silnika parowego, elektrycznych obwodów czy wreszcie - komputera. Stąd filozofowie antyczni przyrównywali działanie świata do wrzeciona na krosnach, siedemnastowieczni do wielkiego mechanizmu, zaś niektórzy ze współczesnych postrzegają go jako symulację podobną do działania komputera (Bolter, 1990, pp. 45-78). To jednak nie koniec listy powodów, dla których oprogramowanie jest istotnym tematem badawczym dla szeroko rozumianej humanistyki. Najnowsza *technologia definiująca* wyszła bowiem z ram bycia wygodną konceptualizacją pewnych procesów i stała się w nich aktywnym agentem. Czynnikiem, który w sposób rzeczywisty i wymierny wpływa na sposób funkcjonowania kolejnych warstw między innymi kultury i sztuki. Oprogramowanie jest w świecie metafor *napędzających* działanie zjawisk będących przedmiotem zainteresowania humanistyki nową jakością. Stanowi bowiem, jak to zostało na kartach poniższej dysertacji wykazane, ideę ucieleśnioną. Jego kluczowa charakterystyka wyraża się w tym, że będąc tworem konceptualnym realizuje się wyłącznie w konkretnym działaniu. "Za sprawą oprogramowania nie tylko pojawiły się nowe obiekty kultury, ale zmieniła się też logika funkcjonowania ich poprzedników. Języki programowania w pewnym sensie wyznaczają granice naszego świata." (Filiciak, 2008, p. 15)

Podstawową motywacją przeprowadzonych rozważań jest próba przyjrzenia się procesom zachodzącym w wyniku powszechnej komputeryzacji (a właściwie: *softwaryzacji)* sposobów tworzenia, przetwarzania i dystrybuowania obiektów kultury i sztuki w *erze cyfrowej.* Celem dysertacji jest wykazanie w jaki sposób działanie oprogramowania, ugruntowane w jego bardzo specyficznej konstrukcji i podstawie teoretycznej wpływa na całą gamę procesów będących w obszarze zainteresowania w pierwszej kolejności nauk o sztuce, następnie zaś kulturoznawstwa i medioznawstwa, a także filozofii (zgodnie z wykształceniem i kompetencjami autora pracy).

Prezentowana praca składa się z czterech części oraz załączników i bibliografii.

Pierwsza część pt. *System* zawiera elementy niezbędne do refleksji nad sposobami działania paradygmatu *softwarowego* w badaniach nad zmediatyzowaną sztuką i kulturą. Pojawiają się tu podstawowe pojęcia i idee leżące u podstaw działania oprogramowania: Maszyna Turinga, dane i informacja, komputacja i procesy z nią związane. W pierwszej części następuje również refleksja nad specyficznym, językowym charakterem oprogramowania i jego niektórymi historycznymi korzeniami. Wyjaśnione zostaje pojęcie algorytmu i prezentowane są przykłady praktyk artystycznych o podłożu algorytmicznym.

Druga część, nosząca tytuł *Analityka* to najobszerniejszy fragment dysertacji, który od rozważań kwestii fundamentalnych dla działania oprogramowania przechodzi do analizy konkretnych sposobów jego funkcjonowania. Następuje więc wyjaśnienie czym jest *software* i jak jest zbudowany. Rozdziałom poświęconym strukturom danych i algorytmom towarzyszą przykłady paradygmatycznych dzieł sztuki nowych mediów, które czynią użytek z omawianych zagadnień. Znaczna ilość uwagi została poświęcona zjawisku *demosceny* jako rodzajowi sztuki komputerowej najbardziej bezpośrednio związanemu z medium. W rozdziale trzecim drugiej części następuje również teoretyczne uzasadnienie dla włączenia aktywności związanych z tworzeniem *softwaru* - programowania - w zakres dziedziny kreatywnej, a nawet artystycznej działalności człowieka, co znów zostaje poparte pewnymi przykładami z dziedziny sztuki nowych mediów. W dalszym toku *Analityki* uwaga zostaje skupiona na zjawiskach związanych z *softwarem* jako metaforą, którą *żywią się* zróżnicowane sposoby myślenia i tryby funkcjonowania idei. Stąd konieczność krytycznego spojrzenia na kod (*Critical Code Studies)* między innymi przez pryzmat specyficznego rodzaju poezji, która wykorzystuje składnię współczesnych języków programowania. Jednym z najważniejszych pojęć, które z języka humanistyki

zostały zapożyczone do informatyki jest "pamięć" - kwestiom związanym z tym przejściem poświęcony jest rozdział siódmy drugiej części pracy. Rozważania *Analityczne* zamyka refleksja na temat nowego paradygmatu technologicznego jakim jest *przetwarzanie rozpowszechnione* (ubiquitous computing) w połączeniu z *Internetem rzeczy* (Internet of Things). Jest to zagadnienie o tyle ważne, że wprowadza do naszego myślenia o oprogramowaniu nową domenę ontologiczną - aktywne przedmioty w kontekstach (artystycznych, społecznych, przestrzennych) i semantyczne przetwarzanie informacji. Część drugą pracy zamyka krótka refleksja nad esejem Friedricha Kittlera *There is no software* (vide: Kittler, 1997)

Część trzecia nosi, znów zaczerpniętą od jednej z kategorii filozoficznych rodem z antycznej Grecji, nazwę *Dynamiki.* Kluczem do prowadzonych tu rozważań jest formowanie się na bazie oprogramowania relacyjnych struktur wymiany danych i informacji. Pojawia się tu istotna kategoria *protokołu* oraz - zaczerpnięte z nauk ścisłych - pojęcie *transdukcji.* Jednym z najważniejszych aspektów informacyjnej *Dynamiki* jest oczywiście *partycypacyjność.* Całość jest ilustrowana przykładami przedsięwzięć o charakterze artystycznym działających na bazie opisywanych *Dynamik* i ujawniających ich zakrytą konstrukcję: net artem, działaniami opartymi na mediach lokacyjnych czy poszerzonej rzeczywistości.

Ostatnia część - *Techne* stanowi swojego rodzaju rozszerzenie dla części poprzednich. Jedną z głównych myśli, przewijających się przez karty poniższej dysertacji jest teza o dynamicznym, partycypacyjnym charakterze mediów softwarowych. Stąd też wniosek, że bardzo często najlepszym sposobem prowadzenia refleksji nad opisywanymi fenomenami jest wejście z nimi w czynną, czasem nawet krytyczną interakcję. Stąd ostatnia część pracy stanowi prezentację wybranych narzędzi, za pomocą których można w przestrzeni akademickiego *Media Labu* czy *Hackerspace* prowadzić działania niosące jednocześnie wartość dydaktyczną jak i badawczą. Do wymienionych narzędzi należą np. język programowania Processing, czy platformy *open hardware* takie jak Arduino czy Raspberry Pi.

Współczesna rzeczywistość komunikacyjna nie pozostawia nam wyboru. Musimy nauczyć się *rozumieć* kod. Nie musi to od razu oznaczać stania się wykwalifikowanym programistą. Wystarczy zdawać sobie sprawę z tego *jak działa* oprogramowanie.

Część 1: System (Σύστημα).

Σύστημα. System. „Termin oznacza organicznie zbudowaną całość. Na płaszczyźnie ontologicznej stoicy np. definiowali kosmos jako »sýstema« nieba, ziemi, wszystkich mieszczących się w nim rzeczy (...) W sensie logiczno-gnozeologicznym przez system rozumie się »prawdę, będącą zespołem wiedzy o rzeczach prawdziwych« (por. Sekstus Empiryk, Zarysy pirrońskie, II, 84, A. Krokiewicz). – Ogólnie systemem nazywa się całość myśli jakiegoś filozofa, o ile jest ona utworzona z części organicznie ze sobą powiązanych, które zależą od jednej lub kilku podstawowych zasad. Myśl filozofów jest bardziej lub mniej systematyczna w zależności od tego, czy wspomniane powiązania są bardziej lub mniej uwypuklone i w sposób zamierzony realizowane. Przeciwieństwem systemu jest to co aporetyczne lub problematyczne, albo – jak to się dziś mówi – otwarte.” (Reale, 2005, p. 227)

Σύστημα 1: Wyznaczanie paradygmatu. Epistemologia jako teoria informacji.

„W zakresie liczb idee są i bardziej sprecyzowane i lepiej dają się odróżniać jedne od drugich aniżeli w zakresie rozciągłości, gdzie nie można tak łatwo jak w liczbach zaobserwować i mierzyć każdej równości i każdego nadmiaru wielkości. Jest tak z tej racji, że nie możemy jeśli idzie o przestrzeń, dojść myślą do pewnej określonej małości, takiej jak jedność liczbowa, poza którą nie moglibyśmy już wyjść”

(Leibniz, 1955, p. 170)

„1. Świat jest wszystkim, co jest faktem.

1.1. Świat jest ogółem faktów, nie rzeczy. (...)

2.012. W logice nic nie jest przypadkowe. Jeśli rzecz *może* wystąpić w stanie rzeczy, to jego możliwość musi już w niej być przesądzona. (...)

2.1. Tworzymy sobie obrazy faktów.

2.11. Obraz przedstawia pewną sytuację w przestrzeni logicznej – istnienie i nieistnienie stanów rzeczy.

2.12. Obraz jest modelem rzeczywistości.”

(Wittgenstein, 2000, p. 7)

„Inaczej mówiąc, normalne, hilbertowskie spojrzenie na matematykę mówi, że wszelkie matematyczne prawdy, nieskończona ilość prawd, może być skompresowana do skończonej liczby aksjomatów. Ale istnieje nieskończona ilość matematycznych prawd, które nie mogą być zredukowane wcale, ani o jeden bit!”

(Chaitin, 2005a, p. 8)

Σ 1.1 Epistemologia.

„The Blackwell Dictionary of Western Philosophy” definiuje „epistemologię”, pojęcie wywodzące się od greckich *episteme* i *logos* jako odnoszące się do naukowej refleksji nad kwestiami związanymi z pozyskiwaniem, uzasadnianiem i naturą wiedzy (Nicholas Bunnin, 2004, p. 218). Podstawowym problemem dziedziny filozofii określanej tym mianem jest między innymi analiza kryteriów, według których pewne orzeczenia uznawane są za prawomocne. Sięga więc epistemologia swoimi korzeniami do jednej z fundamentalnych kwestii filozoficznych – problemu prawdy i prawdziwości naszych wypowiedzi o świecie. W swojej nowoczesnej wersji nauka ta koncentruje się więc przede wszystkim na rozróżnieniu między wiedzą w sensie ścisłym, a ogólnymi przekonaniami (czasem definiując tą pierwszą właśnie jako „dobrze uzasadnione przekonanie”). Sformułowanie współczesnych prawideł związanych z przetwarzaniem i dystrybuowaniem informacji cyfrowej stanowi dla epistemologii krok o ogromnym znaczeniu. Oto bowiem obok ściśle teoretycznych rozważań na temat sposobów pozyskiwania i weryfikowania pewnych twierdzeń na temat świata objawiły się konkretne metody ilościowego ujmowania informacji jako podstawy dla procesów poznawczych. Niedługo po nich stworzone zostały urządzenia, które stanowiły bezpośrednią implementację opracowanych, głównie na gruncie matematyki i logiki twierdzeń. Maszyny te wprowadzane do powszechnego użytku od połowy XX wieku zrewolucjonizowały nasz sposób produkcji, przetwarzania i dystrybucji całych klas zjawisk związanych z szeroko rozumianym pojęciem wiedzy – w tym fenomenów o charakterze kulturowym i artystycznym.

Nie dajmy się zwieść pozornej rozdzielności kwestii epistemologicznych, metafizycznych (czy ontologicznych) z refleksją należącą do dziedziny kulturoznawstwa, nauk o sztuce czy społeczeństwie. Aby podjąć próbę zrozumienia mechanizmów regulujących współczesne obiegi komunikacyjne musimy przyjrzeć się ich filozoficznym podstawom – ze szczególnym uwzględnieniem tych, które dają się zoperacjonalizować tak, aby było możliwe ich przełożenie na język zrozumiały dla działających w oparciu o

konkretne prawa fizyczne maszyny. Od abstrakcji metafizycznej na temat funkcjonowania mechanizmów poznawczych (z fundamentalną kwestią prawdy i prawdziwości) przejść musimy do refleksji nad możliwymi sposobami czynienia jej intersubiektywną, kwantyfikowalną i przetwarzalną. Uczynimy to, by na końcu przekonać się jak wyglądają empiryczne, już działające mechanizmy regulujące produkcję, obróbkę i dystrybucję tego, co wiemy i myślimy na temat świata. Stąd niełatwa droga nasza wiodła będzie od epistemologii, przez teorię informacji do kwestii związanych z oprogramowaniem i jego materialną podstawą – technologiami komputerowymi.

Σ 1.2 Prawda; Prawdy, Prawa.

Nie ma niczego bardziej dla nas naturalnego niż poczucie bycia w harmonii z Uniwersum. Działając na co dzień w Świecie przekonani jesteśmy, że jest on zasadniczo taki, jakim go postrzegamy. Co więcej – poddaje się on z dużą plastycznością naszemu działaniu. Stąd tak zwana *klasyczna definicja prawdy* nieczęsto budzi nasz niepokój. Kantowska „zgodność poznania ze swoim przedmiotem" (Kant, 1957, p. 146) wydaje się być powszechnie najlepiej ukonstytuowanym paradygmatem postrzegania rzeczywistości. Nie mniej jednak tradycja oddzielania bytu (tego co istnieje) od zjawiska (tego co postrzegamy) posiada potężne ugruntowanie w filozofii – od starożytnych Parmenidesa czy Platona, przez Leibniza i cały zbiór filozofów racjonalistycznych (czyli tych, którzy skłonni są wierzyć rozumowi raczej niż oczom) po współczesnych mędrców z całym zróżnicowaniem ich refleksji. Dla myślicieli, szczególnie od czasów nowożytnych dualizm istniejącego i postrzeganego był źródłem intelektualnego niepokoju – łagodzonego za pomocą czy to radykalnego sceptycyzmu (Kartezjusz, Hume), paralelizmów (Leibniz) albo nieco mniej w tym miejscu interesującego naturalizmu (Hobbes). Fundamentalnym jednak osiągnięciem filozofii metafizycznej czasów zamierzchłych wydaje się powiązanie metodologiczne rozważań nad bytem z logiką i matematyką jako naukami po pierwsze traktującymi na temat Prawd (Leibniz użyłby wyrażenia „Praw", co będzie istotne później), po drugie - swoją obiektywnością i abstrakcyjnością dającymi aplikować się bez potrzeby bezpośredniego odniesienia do partykularnych przypadków w świecie.

Σ 1.3 Logika i matematyka. W poszukiwaniu fundamentu.

Z punktu widzenia tematyki niniejszej pracy najbardziej interesujące przemiany w logice i matematyce miały miejsce pod koniec XIX i w pierwszej połowie XX wieku. Rewolucja jaka wtedy zaszła w naukach ścisłych przyniosła bowiem plon w postaci formalizacji, które umożliwiły realizację ogromnego marzenia racjonalistów – skonstruowania maszyny do przetwarzania wiedzy. Stało się to możliwe w wyniku ogromnego trudu przebudowania systemu logiki jaki podjęli wybitni myśliciele tamtych czasów – Bertrand Russell, Alfred North Whitehead, Kurt Gödel, Gottlob Frege i inni. Ich działania wynikające z głębokiego sceptycyzmu i krytyki wobec zastanych prawideł rozumowania umożliwiły w sposób bezpośredni implementację praktyk, które nie mogły powieść się ich szlachetnym poprzednikom kilkaset lat wcześniej. Musimy sobie uświadomić, że pod koniec dziewiętnastego stulecia matematyka znalazła się w sytuacji, którą Russell zwykł przyrównywać do wspaniałego gmachu opartego na zgniłych fundamentach, który pomimo swojego majestatu musi ostatecznie runąć. Podobny problem trawił – zdaniem myśliciela – również filozofię. Podzielał on ideał Leibniza (rzecz warta podkreślenia!) wyrażany w okrzyku „Calculemus!" – „Obliczmy to!". Zbuntowanemu matematykowi marzył się system oparty na stabilnych podstawach i logicznie niewzruszalnym języku – tak, aby dawało się z niego wyciągać jedynie słuszne epistemicznie (niezbicie prawdziwe) wnioski. Aby mógł ziścić się leibniziański sen o *calculus ratiocinator* najpierw musiały zostać przebudowane podstawy wszelkiego rozumowania. Z filozofii ostać się mogło jedynie to, co da się sprowadzić do zestawu nieredukowalnych prawd logicznych – aksjomatów. Najlepiej jeśli dałoby się je zapisać w postaci prostego systemu symbolicznego - co było nie do końca nowym pomysłem, lecz w tym wypadku istotą sprawy stało się uczynienie nowego rodzaju zapisu pierwotnym a nie pomocniczym w stosunku do rozważań prowadzonych w języku naturalnym. Projekt Russella podjęty razem z Whiteheadem miał na celu krytyczne przyjrzenie się dotychczasowym osiągnięciom w dziedzinie logiki i sprawdzenie czy uda się skonstruować z nich fundament wiedzy dostatecznie stabilny aby utrzymać chwiejący się gmach matematyki.

W trakcie realizacji swojego zamierzenia filozofowie natrafili na licznych sojuszników w poszukiwaniach dającej się ująć Prawdy. Należał do nich Gottlob Frege, który już wcześniej podejmował próby opracowania logicznego języka, nadającego się nie

tyle do przeprowadzania prostych mechanicznych operacji na wyrażeniach lecz do modelowania całej rzeczywistości. Uczynił to m.in. w ramach swojego konceptu *Begriffschrift*. Rzecz polegała na próbie zastosowania rachunku logicznego w duchu Leibniza do zrekonstruowania podstaw matematyki (jakże blisko pomysłów Russella!). Rachunek Fregego, od którego często wyznacza się nowy rozdział w historii logiki zawiera między innymi elementy dwuwartościowej logiki predykatów drugiego rzędu z równością i zmiennych kwantyfikowanych. Filozof posługiwał się specyficznym zapisem polegającym na użyciu konektorów i kwantyfikatorów w postaci linii łączących formuły, który jednak daje się stosunkowo łatwo przełożyć na notację współczesną z linearnymi znakami negacji, koniunkcji czy kwantyfikacji. Frege zaproponował dziewięć wyrażeń aksjomatycznych, które jego zdaniem miały charakter samooczywistych prawd. „Jeśli zadaniem filozofii jest przełamanie dominacji słów w ludzkim umyśle (...) moja notacja, opracowana dla tego celu może być użytecznym narzędziem dla filozofów" – pisał (Frege, 1979, p. VI). Pośród innych tropów, którymi podążali Russell z Whiteheadem była m.in. teoria mnogości Georga Cantora, narzędzie które pozwoliło nieco później na ważny sposób konceptualizowania obiektów matematycznych takich jak funkcje czy nawet same liczby (0 jako zbiór pusty, 1 jako zbiór zawierający zbiór pusty, 2 jako zbiór którego elementem jest zbiór pusty oraz 1 itd.). Do zakodowania struktur matematycznych potrzebujemy więc zasadniczo zbioru pustego i grupy aksjomatów. Stanowiło to dobry „punkt zahaczenia" dla logicznych podstaw matematyki.

W 1900 roku na Międzynarodowym Kongresie Matematyków David Hilbert sformułował grupę problemów, których rozwiązanie stanowiło istotną siłę napędową dla rozwoju XX wiecznej matematyki. Niedługo później chęć usunięcia sprzeczności odnalezionych w rozważaniach Fregego (w tym słynnego Paradoksu Russella) pchnęła ostatecznie lorda Bertranda i jego przyjaciela Alfreda Northa Whiteheada do rozpoczęcia prac nad wielką syntezą o nazwie „Principia Mathematica", finalną próbą planowanej od dawna przebudowy matematyki na język logiki. Projekt nie został ostatecznie zamknięty, powstały trzy z planowanych czterech tomów. W 1928 roku w czasie jednego z kolejnych Kongresów Matematyków Hilbert ponownie postawił część ze swoich problemów, czyniąc jednak niektóre z nich bardziej precyzyjnymi. Trzy najważniejsze dają się sprowadzić do pytań: „Czy matematyka jest kompletna?", „Czy matematyka jest niesprzeczna?" i wreszcie „Czy matematyka jest rozstrzygalna?". Dwa z nich zostały rozwiązane w 1930 roku przez Kurta Gödla m.in. w słynnym twierdzeniu o niezupełności, które dowodzi, że w aksjomatycznej, niesprzecznej teorii matematycznej, zawierającej pojęcie liczb

naturalnych da się sformułować takie zdanie, którego w granicach tej teorii nie da się zweryfikować ani sfalsyfikować. Dowód ten zakończył próby aksjomatyzacji matematyki wykazując niewykonalność tego zadania. Trzecia kwestia, znana pod nazwą *Entscheidungsproblem* miała swoje źródło w pomyśle Leibniza (sic!) związanym z jego próbami skonstruowania urządzenia do manipulacji symbolami w celu wykazania prawdziwości orzeczeń matematycznych. Zdawał on sobie sprawę z potrzeby opracowania specjalnego języka, który będzie pozwalał na formalizację problematycznych zagadnień. *Entscheidungsproblem* dotyczy tego, czy jest możliwe zbudowanie receptury (algorytmu), która w skończonej liczbie kroków rozstrzygałby czy dane wyrażenie (formuła) logiki pierwszego rzędu jest czy nie twierdzeniem tej logiki. Ze względu na to, że Gödel wykazał wcześniej pełność tej logiki w rzeczywistości pytamy o twierdzenie jako o tautologię. Zadanie zostało rozwiązane sześć lat później przez Alonzo Churcha lecz ostatecznej (oraz bardziej eleganckiej) konkluzji w tym zakresie dostarczył młody student King's College w Cambridge Alan Turing, który zaproponował niezwykły rodzaj dowodu w postaci teoretycznej maszyny.

Σ 1.4 Maszyna Turinga. Logika w działaniu.

Abstrakcyjne urządzenie, nazwane od imienia swojego twórcy Maszyną Turinga składa się (hipotetycznie) z nieskończonej w obu kierunkach taśmy zawierającej pola oraz „głowicy". Na każdym polu może zostać zapisany jeden z symboli (których liczba musi być jednak skończona). Po taśmie porusza się „głowica", która ma zdolność przechowywania pewnych stanów (których wybór – znów – musi być skończony). W danym momencie może się ona znajdować tylko nad jednym polem. Po odczytaniu zeń symbolu może wykonać jedno z działań:

1. Zatrzymać się (stop),
2. Przemieścić się o jedno pole w dowolnym kierunku,
3. Wpisać dowolny symbol w zajmowane przez inny symbol pole.

Opis aktywności maszyny może zostać opisany jej grafem i składa się z następujących czynności:

a. Podanie aktualnego stanu maszyny,
b. Podanie symbolu aktualnie odczytywanego przez „głowicę",

c. Podanie operacji, którą wykona maszyna zgodnie ze swoim „programem”,

d. Podanie następnego stanu maszyny (po wykonaniu c.).

Inaczej maszynę Turinga możemy opisać za pomocą krotki: $M =< Q, \Sigma, \delta, \Gamma, q_0, B, F >$

Gdzie:

Q – to skończony zbiór stanów,

Σ – to zbiór symboli wejściowych,

δ – funkcja odczytująca aktualny stan i symbol wejściowy,
zwracająca symbol jaki ma się pojawić na taśmie, kierunek przesunięcia głowicy oraz następny stan maszyny,

Γ – skończony zbiór możliwych symboli,

q_0 – stan wyjściowy,

B – symbol pusty,

F – zbiór stanów końcowych.

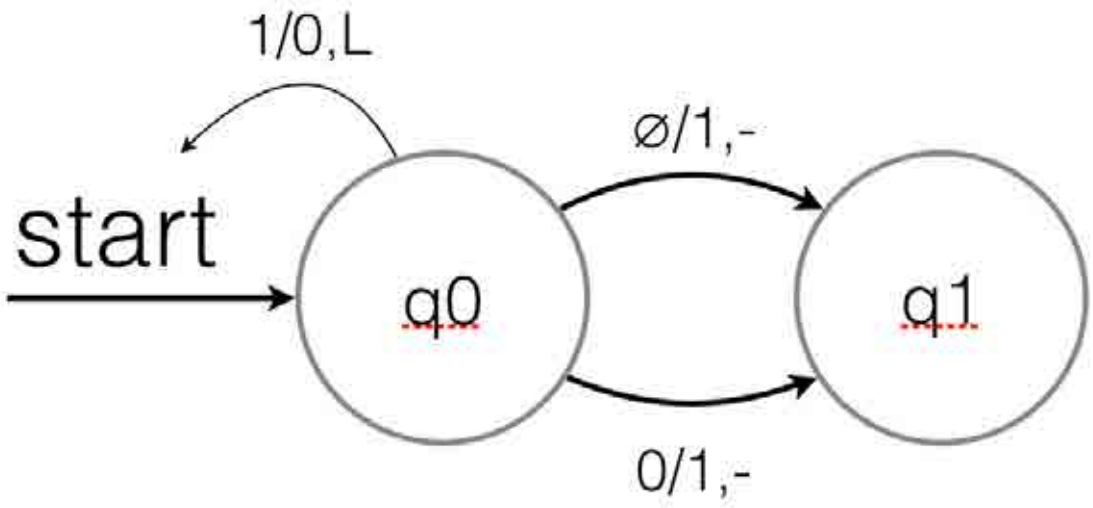
Tak określone abstrakcyjne „urządzenie” posłużyło Turingowi do wykazania, że nie istnieje efektywna, zapisywalna jako algorytm procedura, która użyta na dowolnej Maszynie w skończonej ilości kroków pozwoli na udzielenie odpowiedzi na pytanie: „czy Maszyna zostanie kiedykolwiek zatrzymana (zakończy program)?”. Kwestia ta nosi nazwę *Problemu Stopu (Halting Problem),* a sukcesem Turinga było wykazanie jego nierozstrzygalności.

Uogólniając: dla każdego algorytmu możemy zadać pytanie czy realizujący go program zatrzyma się w skończonym czasie. Możemy tą kwestię sformułować dla konkretnych danych wyjściowych albo dla wszelkich możliwych. Program, który zatrzymuje się bez względu na dane *ma własność stopu.* Istotne jest, że nie istnieje uniwersalny algorytm rozstrzygający problem stopu dla wszystkich algorytmów. Powyższe kwestie mają fundamentalne znaczenie dla informatyki, a także – pośrednio – dla całej filozofii nauki czy epistemologii. Docieramy tutaj bowiem do wyrażonej w 1943 roku przez Stephena Kleene hipotezy nazwanej imieniem ludzi, którzy umożliwili jej powstanie *Tezą Churcha-Turinga*. Ma ona wiele sformułowań ale jej ogólny sens jest następujący: „każdy problem, który jest fizycznie obliczalny, może zostać rozwiązany za pomocą Maszyny Turinga”, czasami mówi się też o tym, że „każdy ‘rozsądny’ model obliczeń może być

efektywnie zasymulowany na Maszynie Turinga". Niezwykłość całego konceptu Maszyny polega na tym, że nie jest ona konkretną formułą, wzorem, twierdzeniem które dałoby się udowodnić bądź obalić. Brytyjski matematyk zaproponował rozwiązanie subtelnych kwestii poprzez zaaplikowanie pewnego rozumowania, dynamicznego tworu, konceptu, eksperymentu logicznego o dużym stopniu precyzji – nie tak wysokim jednak aby dało się matematycznie sprawdzać i uogólnić dla wszystkich możliwych przypadków.

Maszyna Turinga (albo w przypadku jej ogólnego sformułowania Uniwersalna Maszyna Turinga) jest niezwykłym zwieńczeniem i świadectwem zmagań jakie toczyła logika i matematyka początków XX wieku z sobą samą. Skutki tego, że odpowiedź na *Entscheidungsproblem* została udzielona w taki, a nie inny sposób są niewymierne. Oto bowiem skonstruowany został abstrakcyjny model komputera – maszyny służącej do wykonywania algorytmów. Jest to szczególny przypadek w historii kiedy kariera wynalazku zaczęła się od całkowicie abstrakcyjnego wyjaśnienia istoty jego działania. Praktycznym przybliżeniem UMT są wszystkie nasze współczesne komputery. „Wynalazek" Brytyjczyka charakteryzuje się bowiem ogromną podatnością na modyfikacje – da się więc np. wykazać, że maszyny wielotaśmowe są równoważne maszynie jednotaśmowej etc. Dzisiejsza wiedza nie pozwala na jednoznaczne stwierdzenie czy w świetle obowiązujących praw fizyki da się skonstruować modele (czy znów „maszyny") obliczeniowe silniejsze niż Maszyna Turinga co oznacza, że najpotężniejsze współczesne superkomputery ze złożonymi językami programowania są „tej samej wartości" co najprostszy komputer wyposażony w prosty język. Różnica leży oczywiście w czasie wykonywania zadań, jednak przy założeniu nieskończonego czasu są w stanie rozwiązać wyłącznie te same problemy – dzielą możliwości jak i ograniczenia.

Na marginesie : przykład prostej Maszyny Turinga (graf przejść i tabela stanów).



Jest to maszyna dodająca 1 do liczby binarnej (w zapisie dwójkowym).

	q_0	q_1
∅	q_1 1,-	SK
0	q_1 1,-	SK
1	q_0 0,L	SK

Σ 1.5 Epistemologia jako teoria informacji.

Punktem wyjścia dla niniejszych rozważań był projekt Russela dotyczący przebudowania fundamentów gmachu matematyki. Napotkał on na swojej drodze licznych towarzyszy – takich, którzy dołączyli do niego w poszukiwaniach Prawdy matematycznej oraz takich, którzy już znaleźli się na tej drodze w przeszłości. Przyświecała im (czasem uświadomiona, czasem nie) Leibniza koncepcja, wedle której Bóg konstruując świat jednocześnie maksymalizuje jego różnorodność i minimalizuje liczbę idei tworzących jego konceptualną złożoność. Twierdzenie o tym, że żyjemy w najlepszym ze światów oznacza

dokładnie tyle, że jest on doskonale „ekonomiczny" pod względem praw go opisujących. Świat, który zawierałby większą liczbę reguł byłby „nadmiarowy" i przez tą redundancję niedoskonały. Świat bardziej prosty – zdaniem filozofa – nie mógłby być w ogóle. Pomimo ateistycznego światopoglądu Russella (jest on czasem uznawany za „patrona" ateizmu i pacyfizmu jednocześnie) możemy być pewni, że zgodziłby się on co do tak postawionej kwestii natury Uniwersum. Jeśli wrócimy myślą do wątku epistemologicznego, od którego pozwoliłem sobie zacząć ten rozdział, będziemy chcieli zadać pytanie o status teorii poznania w świetle poczynionych dywagacji. Skąd możemy wiedzieć, że nasza wiedza jest prawdziwa? Jak możemy zmierzyć skuteczność teorii i odróżnić ją od takiej, która nie działa lub jest nadmiarowa (grzech w oczach logiki!)?. Rozwiązanie podsuwa nam argentyńsko-amerykański matematyk i informatyk Gregory Chaitin na gruncie swojej *algorytmicznej teorii informacji*. Pisze on:

„Epistemologiczny model centralny dla tej teorii polega na tym, że naukowa lub matematyczna teoria jest traktowana jako program komputerowy służący do obliczania faktów. Im mniejszy program tym lepiej. Złożoność teorii, naszych praw (prawd – dop. J.K.A.) jest mierzona w bitach oprogramowania:

Program (ciąg bitów) → **Komputer** (Maszyna Turinga – dop. J.K.A.) → Wynik (ciąg bitów)
Teoria → **Komputer** → Matematyczne lub naukowe fakty
Rozumienie jest kompresją!"

(Chaitin, 2005a)

Uniwersalna Maszyna Turinga, komputer – i wprowadzona (a w zasadzie uściślona) przez nią koncepcja obliczalności i obliczeniowości (*computation*) nie stanowi jednoznacznego rozwiązania problemów epistemologii. Otwiera nas jednak na nową dziedzinę wiedzy, która w synergii z nowym, wspaniałym wynalazkiem uruchamia pole do dalszych, tym razem bardziej konkluzywnych rozważań. Tą dziedziną jest **teoria informacji**, której dokładniejszej analizie poświęcony jest drugi rozdział niniejszej dysertacji. Pozwala ona nie tylko zająć się procesami kwantyfikowania, przetwarzania i optymalizacji wiedzy ale pozwala na szerokie i precyzyjne włączenie w zakres naszego zainteresowania fenomenów o charakterze językowym (tekstualnym). Chcąc poddać refleksji sposoby na jakie oprogramowanie wpływa na tworzenie, przetwarzanie i

dystrybuowanie fenomenów o charakterze kulturowym i artystycznym (i w drugą stronę tzn. w jaki sposób szeroko rozumiana kultura i sztuka kształtuje technologie informatyczne) potrzebujemy intelektualnego interfejsu, który pozwoli nam połączyć te dwa światy. Dynamika układu wyznaczana jest za pomocą trójki:

Maszyna Turinga (jako model obliczeniowości komputerowej) ↔ Dane (teksty, zjawiska kultury) ↔ Język (rozumiany jako system/kod i specyficznie nasze, ludzkie narzędzie dostępu do i konceptualizacji świata).

Σύστημα 2: Informacja, dane, wiedza. Kod i struktura.

„Ze względu na szeroki zakres obejmowany tematem komunikacji, da się wyróżnić problemy na trzech poziomach. Dlatego wydaje się właściwym aby zapytać kolejno o:

POZIOM A.
Jak dokładnie mogą być transmitowane symbole w komunikacji?
(Problem techniczny).

POZIOM B.
Jak precyzyjnie transmitowane symbole pokrywają pożądane znaczenie? (Problem semantyczny).

POZIOM C.
Jak efektywnie odebrane znaczenia wpływają na zachowanie w pożądany sposób? (Problem efektywności)”

(Claude E. Shannon, 1980, p. 4)

„Jednym z istotniejszych problemów teorii informacji jest ustalenie miary ilości informacji. Można wymienić trzy metody określania ich miary, a mianowicie:

- podejście uwzględniające strukturalną budowę informacji,
- podejście uwzględniające jej zależności statystyczne,
- podejście uwzględniające semantyczną wartość informacji.”

(Kuriata, 2001, p. 11)

„Informacja: znaczenie przypisane pojedynczemu elementowi danych zgodnie ze znanym umownym systemem. Przykład: ‘2’ jest pojedynczym elementem danych; gdy zdefiniujemy ‘2’ jako symbol lutego, drugiego miesiąca roku, ‘2’ staje się informacją”

P.E. Butron (za: Ifrah, 2006, p. 860)

Σ 2.1 Problem pojęcia „informacji”.

Istnieje ogromna ilość dostępnych definicji informacji. Praktycznie każda czyniąca z tego pojęcia użytek dziedzina wiedzy posiada własne jego sformułowanie. Może informacja być traktowana w ujęciu zbliżonym do potocznego, kiedy na myśli mamy coś, co poszerza nasze rozumienie: „informacja: wiadomość, przekaz: coś, co dostarcza nam wiedzy” (Le Grand Livre des techniques, Paryż 1978)[1]; zestaw danych: „informacja: zbiór podstawowych danych dostarczanych przez środowisko istocie żywej, a zwłaszcza człowiekowi (wrażenia) lub niektórym maszynom” (H. Piéron); jedno z kluczowych pojęć informatyki i cybernetyki: „informacja: zrozumiałe dane dowolnego rodzaju. W szczególności pojęcie to obejmuje wszelkie sformułowania abstrakcyjne, które mogą odzwierciedlać i opisywać pewien element – sytuację, cechę, wydarzenie – dowolnego systemu posiadającego strukturę” (J. Bureau); czy też koncept należący do abstrakcyjnych rozważań z zakresu matematyki: „informacja zawarta w systemie stanowi stosunek liczby odpowiedzi, jakich system mógłby udzielić przed otrzymaniem informacji, do liczby możliwych odpowiedzi po otrzymaniu informacji” (G. Verroust). Tego rodzaju sformułowania można by mnożyć. Najbardziej jednak interesujący wydaje się fakt, że większość z nich wydaje się być prawdziwa i pomimo pozornej, wzajemnej sprzeczności da się skutecznie zoperacjonalizować.

Samo słowo „informacja” pochodzi od łacińskiego *informare* oznaczającego „tworzenie”, „wyjaśnianie” ale też „nadawanie kształtu”. To ostatnie znaczenie wydaje się szczególnie frapujące: *in formatio* – nadawanie formy, struktury: ustanawianie. Źródło słowa zdaje się kryć w sobie sens większy i bardziej dla naszych rozważań istotny niż skłonni byśmy wnioskować z siatki współcześnie afiliowanych przy nim znaczeń. „Informacja” pozostaje jednak wyrażeniem - ujmując rzecz ogólnie – o powszechnie zrozumiałej, choć hybrydycznej i ciągle ewoluującej ekstensji. Punktem wyjścia dla naszych rozważań na temat relacji fundowanego przecież na przetwarzaniu informacji oprogramowania i fenomenów o charakterze kulturowym i artystycznym była współczesna, zdeterminowana przemianami w logice i matematyce epistemologia. Stąd przyjąć należy, że pierwsze istotne z naszego punktu widzenia próby ustabilizowania znaczenia omawianego pojęcia jako samodzielnego wyrażenia, posiadającego w miarę jednoznaczne odniesienie pojawiają się w pierwszej połowie XX wieku w pracach

[1] Cytaty w tym akapicie za (Ifrah, 2006, pp. 858-861)

naukowych, które wyszły m.in. spod piór pracowników Bell Laboratories – amerykańskiego pioniera w dziedzinie telekomunikacji. Ich autorami byli m.in. inżynier Harry Nyquist (*Certain Factors Affecting Telegraph Speed,* 1924) oraz Ralph Hartley (*Transmission of Information,* 1925). W tej ostatniej badacz podsumował rozważania brytyjskiego statystyka i genetyka Ronalda Fishera (*Theory of Statistical Estimation,* 1925*)* oraz wspomnianego Nyquista proponując opracowanie ilościowej miary stosowalnej tam, gdzie rozważane są systemy transmitujące informacje. Dodatkowo poczynił on jedno z kluczowych dla teorii informacji rozróżnień między kwestiami fizykalnymi a psychologicznymi związanymi z przesyłaniem komunikatu. Krótko mówiąc – czym innym było dla niego znaczenie, a czym innym informacja jako to, co można nadać bądź odebrać. To sformułowanie może się wydać współczesnemu czytelnikowi w pierwszej chwili trywialne w swej prostocie, stało się ono jednak podstawą, na której wiele lat później skonstruował swoją teorię Claude Shannon – ojciec nowoczesnej teorii informacji.

Dzisiejsze pojęcie informacji pomimo, że doczekało się szczegółowych analiz, rozwinięć i formalizacji, nadal pozostaje fenomenem o niejednoznacznej konstrukcji i wielobarwnych konotacjach. Shannon starając się sformalizować kwestie związane z opisem zjawisk o charakterze informacyjnym sam wyrażał swoje wątpliwości w kwestii możliwości skonstruowania jednej satysfakcjonującej definicji tej kategorii nawet na gruncie samej tylko teorii informacji:

„Nie należy się spodziewać, że pojedyncza koncepcja informacji mogłaby być z zadowoleniem przyjęta przy tak wielu możliwych polach eksploatacji tego ogólnego zagadnienia.” (Claude E. Shannon, 1980)

Mimo tego Claude Shannon i Warren Weaver podjęli próbę ujęcia najważniejszych kwestii dotyczących pojęcia będącego przedmiotem naszego zainteresowania i w swoim klasycznym modelu wyróżnili trzy poziomy badań nad komunikacją (informacją):

- poziom techniczny (odnośnie kwantyfikacji i transmisji),
- poziom semantyczny (rozpatrujący znaczenie i prawdziwość),
- poziom „cybernetyczny” (dotyczący efektywności wpływu komunikatu),

Z różnych względów najbardziej interesującym dla prowadzonych tu rozważań będzie poziom drugi – dotyczący kwestii związanych z informacją semantyczną i jej przekazywaniem. Stanowi ona bowiem trampolinę do tematyki będącej główną osią niniejszej dysertacji tj. kwestii produkowania, przetwarzania i dystrybucji danych kulturowych w skomputeryzowanej rzeczywistości (za pomocą oprogramowania).

Σ 2.2. Ilościowa miara informacji w teorii komunikacji. Kluczowe pojęcia.

Do najważniejszych pojęć nauki o informacji należą redundancja i entropia. Zacznijmy od ich zdefiniowania.

a) **Entropia** losowej zmiennej x przy prawdopodobieństwie p(x) zajścia zdarzenia x jest definiowana wzorem:

$$H(x) = -\sum p(x)\log_2 p(x)$$

Jeśli używamy logarytmu o podstawie 2 to wynik otrzymamy w *bitach*. Entropia jest więc *miarą uśrednionej niepewności* wystąpienia jakiegoś wydarzenia. W powyższym przypadku jest to – inaczej mówiąc – średnia liczba bitów potrzebna do opisania losowej zmiennej.

W podręczniku *Elements of Information Theory* (Thomas Cover, 1991) znajdziemy następujący przykład:

„Rozważmy losową zmienną o równomiernej dystrybucji z 32 wyników. Aby zidentyfikować wynik, potrzebujemy etykiety, która obejmie 32 różne wartości. Wystarczy nam więc 5 bitowy ciąg. Entropia tej losowej zmiennej wynosi:

$$-\sum_{i=1}^{32} p(i)\log p(i) = \sum_{i=1}^{32} \frac{1}{32}\log = \log 32 = 5bit$$

Możemy też rozważyć przypadek z niejednorodną dystrybucją. Załóżmy, że mamy do czynienia z wyścigiem ośmiu koni. Załóżmy, że prawdopodobieństwa wygranej poszczególnych koni wynoszą:

$$(\frac{1}{2},\frac{1}{4},\frac{1}{8},\frac{1}{16},\frac{1}{64},\frac{1}{64},\frac{1}{64},\frac{1}{64})$$

Możemy obliczyć entropię wyścigu:

$$H(x) = -\frac{1}{2}\log\frac{1}{2} - \frac{1}{4}\log\frac{1}{4} - \frac{1}{8}\log\frac{1}{8} - \frac{1}{16}\log\frac{1}{16} - 4\frac{1}{64}\log\frac{1}{64} = 2bity$$

Jeśli chcemy przesłać komuś informację, który koń zwyciężył w wyścigu, możemy podać jedynie „indeks" konia – będziemy wtedy potrzebowali trzech bitów informacji dla dowolnego z nich. Ponieważ jednak prawdopodobieństwo wygranej każdego ze zwierząt nie jest takie samo - jest sens wybrać dłuższe deskrypcje dla mniej prawdopodobnych zwycięzców, a krótsze dla tych, których wygrana byłaby mniejszym zaskoczeniem. Stąd na przykład możemy kolejne rywalizujące zwierzęta opisać następująco: 0, 01, 110, 1110, 111100, 111101, 111110, 111111. Średnia długość opisu wyniesie – zgodnie z powyższymi rozważaniami – dwa bity, w przeciwieństwie do trzech bitów kodu jednorodnego. (Dodatkowo, w najbardziej prawdopodobnym przypadku wystarczy nam jeden bit opisu: „0" – dop. J.K.A.)"

(Thomas Cover, 1991, p. 5)

b) **Redundancję** możemy zdefiniować w relacji do wyjaśnionego powyżej pojęcia entropii. W kontekście teorii informacji jest ona po prostu naddatkiem informacji w komunikacie (łac. *redundantia* – nadmiar, zbytek) czyli wartością oznaczającą ilość bitów (trzymajmy się tej jednostki jako najprostszej, są oczywiście możliwe inne) użytych do zakodowania informacji minus minimalna ich ilość konieczna do wykonania takiej operacji. Redundancja pełni istotną rolę w systemach telekomunikacyjnych, gdzie nadmiarowa informacja może realizować pozytywną funkcję w przypadku zaszumionych komunikatów, utraty części transmisji itp. Balans między entropią i redundancją jest jednym z podstawowych zagadnień teorii kompresji danych – proces ten ma na celu usunięcie zbędnej redundancji (a więc zmniejszenie objętości zbioru), a jednocześnie zachowanie informacji kluczowych (np. sum kontrolnych dla sprawdzenia czy zbiór nie uległ uszkodzeniu etc.).

c) Skoro już przywołane zostało pojęcie **szumu** – przyjrzyjmy się i jemu. Ten ogólny termin w specyficznym ujęciu teorii informacji oznacza wszelkiego rodzaju niepożądane modyfikacje, które mogą się pojawić w przetwarzanym, transmitowanym, przechowywanym bądź konwertowanym sygnale. Może też oznaczać wszelkiego rodzaju sygnały, które są losowe (nieprzewidywalne – a więc wysoce entropiczne) i nie niosą ze sobą wartości informacyjnej. Szum z reguły traktowany jest jako zbędne obciążenie sygnału, którego nadmiar jest czymś, co należy usuwać. Są jednak specyficzne przypadki

kiedy szum pełni pozytywną rolę i bywa czasem nawet sztucznie generowany na potrzeby danego komunikatu (np. *comfort noise* w komunikacji radiowej i bezprzewodowej).

d) **Sygnał** w interesującym nas paradygmacie jest opisywany jako:

„Funkcja, która przenosi informację na temat zachowania lub właściwości pewnego zjawiska”

(Preimer, 1991, p. 1)

Niesłychanie interesujące jest to, że w świecie fizycznym każdy fenomen przejawiający zróżnicowanie w czasie i w przestrzeni jest potencjalnie sygnałem, który może stanowić źródło informacji na temat stanu fizycznego systemu lub przenosić komunikat między uczestnikami komunikacji. W teorii informacji pod pojęciem sygnału rozumie się jednak przede wszystkim zakodowaną wiadomość. Pewną sekwencję stanów w kanale komunikacyjnym, która koduje pewną informację. Tak jak w przykładzie z wyścigami konnymi mieliśmy do czynienia z informacją w postaci binarnej, która może być przeniesiona za pomocą takich sygnałów jak mignięcia latarką w ciemności, dźwięki gwizdka we mgle lub zmiana napięcia prądu elektrycznego w układzie. Tą informacyjną elastyczność jeśli chodzi o sposób implementacji ilustruje Charles Petzold w książce *Kod. Ukryty język sprzętu komputerowego i oprogramowania* znaną historią z czasów rewolucji amerykańskiej (Petzold, 2002, pp. 81-83). Kiedy Paul Revere ruszał w pośpiechu konno w noc aby ostrzec osadników przed nadciągającymi Brytyjczykami nie wiedział jeszcze, którą z możliwych dróg - morską czy lądową - nadchodzą adwersarze. Nie chcąc jednak tracić czasu ruszył z wiadomością i - wedle opowieści - kazał tym, którzy zostali dać sobie znać o stanie rzeczy za pomocą latarni kiedy już sytuacja się wyjaśni. Jak pisze Henry Wadsworth Longfellow:

„Rzekł przyjacielowi: »Jeśli dziś Anglicy Nadejdą z miasta morzem lub lądem, Zawieś latarnię na łuku dzwonnicy Kościoła północnego. Niech będzie jedna Gdy idą lądem; dwie gdy morzem płyną«” (za: Petzold, 2002, p. 81)

Mamy tu do czynienia z przykładem użycia kodu binarnego zastosowanego do zapisu pewnego rodzaju informacji. Revere chcąc uzyskać precyzyjną wiedzę na temat statusu wojsk inwazyjnych musiał się posłużyć co najmniej dwoma latarniami. Gdyby

dysponował jedną - jedynym komunikatem dającym się przekazać byłaby sytuacja „Brytyjczycy nadchodzą” (latarnia zapalona) lub „Brytyjczycy nie nadchodzą” (latarnia zgaszona), przy czym informacja negatywna byłaby obarczona pewnym stopniem ryzyka - nazwijmy je technicznym. Brak światła latarni może bowiem oznaczać zarówno „Brytyjczycy nie nadchodzą” jak i fakt, że latarnia się zepsuła lub nie udało się skrzesać ognia. Aby uniknąć podobnych wątpliwości często w praktyce stosuje się specjalny rodzaj szumu pełniącego funkcję fatyczną - informującego o utrzymaniu otwartego kanału komunikacji nawet jeśli w danym momencie nie jest transmitowana żadna informacja (podczas blokady Leningradu w czasie II Wojny Światowej w sieci radiowej podczas ciszy informacyjnej transmitowano dźwięk metronomu, który wskazywał, że ten rodzaj komunikacji nadal działa). Wracając jednak do Revera - minimalna ilość bitów, za pomocą której można zakodować interesującą bohatera informację wynosi dwa. Mamy więc następujące możliwe komunikaty:

00 - Brytyjczycy nie nadchodzą
01 - nadchodzą lądem
10 - nadchodzą lądem
11 - nadpływają morzem

(por. ibid.)

Należy zwrócić uwagę, że tu również mamy do czynienia z praktyką mającą na celu wzmocnienie jednoznaczności komunikatu. W ciemności, z daleka nie da się dobrze ustalić czy zapalona jest lampa lewa czy prawa. Nie możemy więc zakodować: 01 jako „nadchodzą lądem”, a 10 jako „nadchodzą morzem”. W tym konkretnym przypadku do zakomunikowania trzech możliwości potrzebujemy aż dwóch bitów informacji. Redundancja wzmacnia jednoznaczność przekazu. Jak widać z powyższego przykładu - dla samej informacji nie ma znaczenia medium, w którym została zakodowana. Revere mógłby użyć wystrzałów armatnich, a za dnia - flag na maszcie itd. Wybór medium wiązał się z konkretną sytuacją - wygodą, bezpieczeństwem, okolicznościami. Nie był zaś konstytuowany możliwością lub niemożliwością zakodowania konkretnej wiadomości.

„I oto, gdy patrzy na wieżę kościoła,
Błysk najpierw, a potem już światło dostrzega.
Wskakuje na siodło, za uzdę już chwyta,
Lecz nadal zwlekając, wpatruje się w wieżę,
By lampę następną wkrótce ujrzeć jasną.”

(cyt. za ibid.)

Stało się. Spośród trzech możliwości została wybrana jedna. Prawdopodobieństwo wystąpienia komunikatu zostaje zamienione na fakt pojawienia się konkretnego sygnału. Informacja zostaje przekazana. Do niesłychanie istotnych dla komputerowego realis kwestii sposobów przechowywania i przetwarzania informacji oraz danych wrócimy w dalszej części pracy.

Pozostańmy jeszcze w uniwersum zero-jedynkowym i przyjrzyjmy się innym właściwościom kodu binarnego. Wiemy już, że:

„Dowolna informacja, która może być sprowadzona do wyboru między dwiema lub wieloma możliwościami, może być wyrażona za pomocą bitów.”

(Ibid, s. 84)

Zagłębiając się w powyższe rozważania, zawędrowaliśmy w pułapkę zapętlenia definicji. Oto bowiem do wyjaśnienia pojęcia sygnału, posłużyliśmy się opisem działania binarnego kodu – ów zaś silnie związany jest z określeniem specyficznej dynamiki entropii i redundancji w układzie komunikacyjnym. Bardzo trudno w systemie tym wskazać pojęcie pierwotne gdyż do czynienia mamy raczej ze elementami, które definiują się wzajemnie. Podejmijmy zatem – skoro już pewne podstawowe fakty zostały nakreślone – próbę usystematyzowania całości zagadnienia.

Σ 2.2.1 Usystematyzowanie pojęć: miary informacji.

Odwołując się do przytoczonego na początku tego rozdziału shannonowsko-weaverowskiego rozróżnienia na trzy poziomy rozważań w dziedzinie studiów nad komunikacją najrozsądniej będzie uściślić pojęcie informacji (i jej miarę) w odniesieniu do trzech typów teorii. Pierwsza z nich, zwana *strukturalną* szczególnie udatnie stosuje się do

interpretacji implementacji w systemach informatycznych, które posiadają konkretną budowę specyficzny sposób kodowania. Interesuje nas tu jak zbudowana jest informacja. Druga, najstarsza w XX-wiecznej genealogii, stanowi odniesienie do procesów *statystycznych* i dotyczy opisywanej już konceptualizacji informacji jako miary nieokreśloności (prawdopodobieństwa) wystąpienia pewnego obiektu, zjawiska, faktu czy sygnału w systemie. Trzecia teoria odwołuje się do pojęcia znaczenia czyniąc swoim przedmiotem przede wszystkim *semantykę* procesów informowania i bycia informowanym. Należy też zwrócić uwagę, że źródła mogą generować sygnał o charakterze dyskretnym, zdeterminowanym z policzalnego zbioru elementów do postaci sekwencji (nazwijmy go tu: cyfrowym) lub swobodniej wyróżnialnym z określonych wielkości fizycznych, których wartość płynnie zmienia się w czasie – mamy wtedy do czynienia ze zjawiskiem ciągłym (analogowym), którego stany możemy wyróżniać przez arbitralne próbkowanie lub kwantyfikację.

a) W przypadku **teorii strukturalnej** mamy do czynienia ze zdeterminowanymi wiadomościami, tworzonymi z zamkniętego zbioru elementów, które są wysyłane sekwencyjnie w czasie. Za Eugeniuszem Kuriatą (Kuriata, 2001) wyróżnić możemy trzy strukturalne metody określania miary informacji: geometryczną, kombinatoryczną i addytywną.

Określenie **geometryczne** polega na pomiarze wielkości obiektu w modelu geometrycznym pewnego kompleksu informacyjnego za pomocą pewnych jednostek, które są odpowiednikami kwantów informacji.

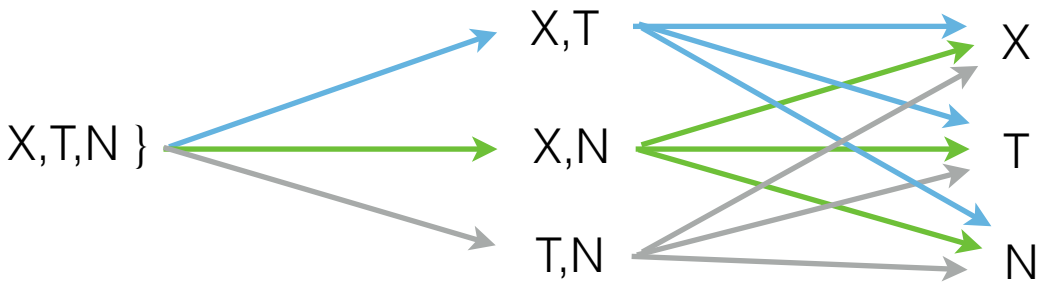
„Niech informacja będzie przedstawiona za pomocą pełnego kompleksu XTN (...) Jeżeli dyskretyzacja będzie odbywać się na osiach X, T oraz N ze skokiem Δ_X, Δ_T oraz Δ_N to ciągłe współrzędne można rozłożyć na elementy (kwanty). Liczbę tych elementów obliczamy ze wzoru:

$$m_x = \frac{X}{\Delta X}; m_T = \frac{T}{\Delta T}; m_N = \frac{N}{\Delta N};$$

W takim przypadku ilość informacji zawartej w kwantach znajdujących się w pełnym kompleksie XTN, określonych wcześniej metodami geometrycznymi będzie równa:
$M = m_X m_T m_N$"

(Kuriata 2001, 13)

Powyższa metoda przybliża sposób dyskretyzacji dla jednakowych osi w układzie stacjonarnym (nie zmieniającym widoku w czasie).



Miara **kombinatoryczna** polega na określeniu zróżnicowania strukturalnych kompleksów informacyjnych. Ma to duże znaczenie dla optymalizacji procesu przesyłania informacji, gdzie możemy uzyskać różne rezultaty stosując alternatywne sekwencje. Zagadnienie to stanie się dla nas interesujące w późniejszej części pracy.

Trzeci ze sposobów mierzenia ilościowej wartości informacyjnej komunikatu wyraża się za pomocą tzw. **miary Hartleya**. Wiemy już, że jeśli mamy do dyspozycji zbiór N-elementowy $X=\{x_1,...,x_N\}$ to wskazanie elementu może polegać na podaniu jego indeksu, alternatywnie możemy też zaaplikować teorię Shannona i opisać element pewną porcją informacji (np. dla systemu binarnego wystarczy nam do tego $\log_2 N$ bitów, w systemie dziesiętnym $\log_{10} N$ jednostek itd.). Analogicznie do powyższego *miara addytywna Hartleya* będzie definiowana:

$$I_{Hartley}(X)=\log N$$

Jeśli więc mamy zbiór NM elementowy $X=\{x_{11},...,x_{1M},..,x_{N1},...,x_{NM}\}$ oraz zbiór N-elementowy $X=\{x_1,..,x_N\}$ to:

$$\begin{aligned} I_{Hartley}(X)&=\log NM=\log N+\log M \\ &=I_{Hartley}(\bar{X})+I_{Hartley}(X_1) \\ &=I_{Hartley}(\bar{X})+I_{Hartley}(XN) \end{aligned}$$

Ilość informacji niezbędna do wyróżnienia konkretnego elementu, nazwijmy go X_{jn} należącego do X może być wyrażona jako suma informacji niezbędnych do

zlokalizowania tego elementu $(I_{Hartley}(X_j))$ o ile wiemy, że znajduje się on w X_j oraz informacji jaka niezbędna jest nam do wskazania zbioru, który jest przedmiotem naszego zainteresowania $(X_j \in \overline{X})$ a więc $I_{Hartley}\overline{X}$. Właściwość tą nazywamy właśnie addytywnością miary informacji Hartleya. Możemy za jej pomocą rozważać sytuacje w której mamy dwa rozdzielne zdarzenia oraz zdarzenie będące ich „przecięciem".

b) Cechą **teorii probabilistycznej** jest nacisk położony na rozpatrywanie sytuacji, w których prawdopodobieństwo pojawienia się pewnej sytuacji nie jest równomierne.

Rozważania tego rodzaju były już przytaczane w punkcie Σ 2.1 tego rozdziału ze względu na aplikację kluczowego dla naszych rozważań pojęcia entropii. Aby jednak trzymać się przyjętego postanowienia o usystematyzowaniu rozważań powtórzmy: kiedy zyskujemy wiedzę o zaistnieniu wysoce prawdopodobnego zjawiska – ilość informacji jaką pozyskaliśmy w procesie komunikacyjnym jest nieduża, jeśli zaś szansa wystąpienia faktu, o którym zostaliśmy poinformowani jest bardzo mała – wiedza nasza jest ilościowo znacznie bardziej wartościowa. W układach zamkniętych rozpatrujemy sytuacje, w których suma prawdopodobieństw poszczególnych wydarzeń składa się do 1. Oznacza to, że jedno z nich zachodzi z całą pewnością. Entropia jako pojęcie jest więc w sytuacji informacyjnej po prostu opisem bogactwa i różnorodności stanów. Shannon jako inżynier zajmujący się przede wszystkim komunikacją zastanawiał się jak zmierzyć ilościowo (w jakich jednostkach) obiektywną „wartość" tego co zostało przesłane w transmiterze i (po uwzględnieniu szumu) odebrane przez odbiorcę. Możliwości było wiele – słowa, sensy, wyrażenia na minutę... wszystkie jednak okazywały się niesatysfakcjonujące ze względu na swoją nieścisłość. Wyjściem okazało się sformułowanie ogólnej, znanej nam już formuły, która pozwalała na określenie – właśnie na bazie probabilistyki – miary na potrzeby komunikacji. Szczególnym przypadkiem jest najprostszy układ, w którym określamy, który z dwóch równie prawdopodobnych stanów zrealizował system – jednostką informacji (wiedzy o fakcie, że „0" lub „1") w takiej sytuacji jest *bit* (**bi**nary dig**it)**. Na cyfrach dwójkowych właśnie operują współcześnie niemal wszystkie urządzenia komputerowe na świecie (nie licząc historycznych i eksperymentalnych projektów opierających się na innych systemach – z naszego punktu widzenia mało interesujących ze względu na niszowy charakter). Głównym powodem powszechnego w świecie IT *(nformation technologies)* liczenia w bitach jest właśnie prostota tego rodzaju operacji,

inne powody będą dla nas interesujące kiedy będziemy zastanawiać się nad kodowaniem informacji oraz implementacją omawianych teorii w systemach elektronicznych.

c) W pierwszej chwili wydawać by się mogło, że **teoria semantyczna** stanowić będzie największy problem jeśli chodzi o przedsięwzięcie wprowadzenia systematycznej miary informacji. Wszak inną rzeczą jest zmierzenie stosunku prawdopodobieństwa wystąpienia pewnego sygnału spośród zbioru możliwych sygnałów, czym innym zaś próba wprowadzenia miary *sensowności* (lub wręcz wartości merytorycznej) tego co zostało przekazane.

Próby rozwiązania problemu mogą polegać na wprowadzeniu miary sensowności utylizującej znane rozwiązanie statystyczne ale poszerzając je o kwestie związane z oceną wartości logicznej wyrażeń informacyjnych (zgodnie z rozumowaniem, że wartość prawdy lub fałszu mogą mieć tylko zdania sensowne). I tak jeśli chcemy zbadać *sensowność* wyrażenia *i* możemy to zrobić konstruując funkcję sensowności *m(i)* lub, wygodniej, stosując negację sensowności wyrażenia:

$$Sens(i) = m(\sim i) = 1 - m(i)$$

Następnie konstruując oparty na prawdopodobieństwie układ zdarzenia $p(i)$ i antyzdarzenia $q(i)$ możemy rozpoznać zbieżność warunków:

$$m(i) + m(\sim i) = 1$$
$$p(i) + q(i) = 1$$

Stąd jeśli statystyczną ocenę ilości informacji określamy zależnością:

$$I = \log_2 \frac{1}{p(i)} = -\log_2 p(i)$$

To podobnie możemy określić logiczną ocenę ilości informacji:

$$Inf = \log_2 (\frac{1}{1 - cont(i)}) = \log_2 \frac{1}{m(i)} = \log_2 m(i)$$

Oczywiście rozwiązanie to nie jest w pełni satysfakcjonujące, nie uwzględnia bowiem pełnej złożoności semantycznej sytuacji komunikacyjnej. Przytoczone formuły

można rozszerzać np. o rozwiązania proponowane podręczniku *Teoria Informacji i Kodowania* (Kuriata, 2001) dodając kwestie związane z miarą zasadności informacji (wpływu na prawdopodobieństwo osiągnięcia celu), jej istotności (z uwzględnieniem relacji w czasie i przestrzeni) czy wreszcie dynamicznej entropii sytuacji.

Ciekawą propozycją, która znajdzie swoje przełożenie na zjawiska związane z działaniem Maszyny Turinga oraz analizą informacji w stylu Shannona jest miara informacji wg. Kołmogorowa. Rzecz zasadza się na bliskim nam koncepcie zastosowania „wynalazku" Turinga do rozwiązania problemu:

„Informację o pewnym obiekcie mamy wtedy, gdy jesteśmy w stanie odtworzyć ten obiekt i jego model na podstawie skończonej liczby cech obiektu (na podstawie jego opisu). Oznacza to, że pełną informację mamy tylko wtedy, gdy jesteśmy w stanie odtworzyć obiekt na podstawie jego opisu".

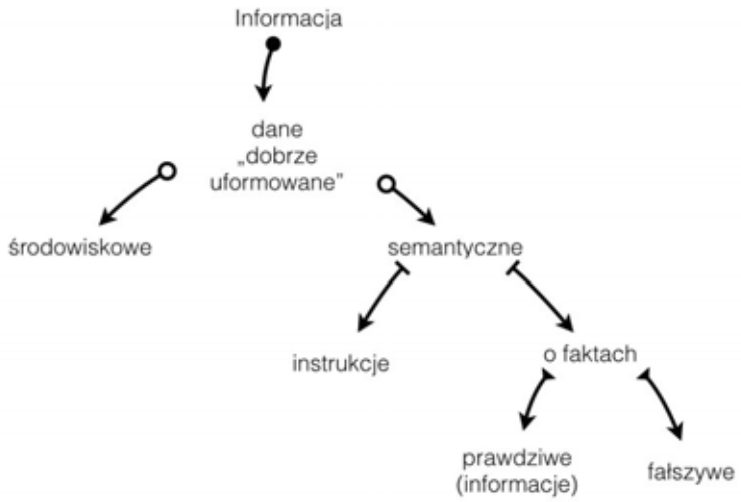
(Kuriata, 2001, p. 71)

Złożoność Kołmogorowa jest definiowana przez funkcję, która mapuje *obiekty* (możemy je rozumieć jako liczby naturalne czy sekwencje symboli, jakie zwrócić nam może zastosowanie modelu Shannona) na liczby naturalne. Mówiąc najprościej - złożoność Kołmogorowa pewnej sekwencji jest wyrażoną w bitach długością najkrótszego programu, który zwraca sekwencję kiedy kończy pracę. Pomimo faktu, że definicja jest wyrażona w odniesieniu do konkretnego modelu maszynowego (MT), z tezy Churcha-Turinga[2] wiemy, że może być ona rozszerzona na inne modele. Złożoność Kołmogorowa obiektu może więc być traktowana jako obiektywne ujęcie ilościowe jego zawartości informacyjnej! W ten oto sposób wkraczamy z teorii informacji w dziedzinę związaną z algorytmiką i świat - tak ważnych dla nas - pojęć z zakresu programowania. Zanim jednak zagłębimy się w pełni w krainę *softwaru* przyjdzie nam jeszcze zastanowić się nad specyfiką implementacji kodów i kodowania (vide: Σ 2A) oraz ich relacjami z językiem (vide: Σ 3).

[2] Istnieje wiele sformułowań tezy np.: „każdy problem, który może być intuicyjnie uznany za obliczalny jest rozwiązywalny przez MT" lub „każda funkcja obliczalna jest rekurencyjnie obliczalna" etc.

Σ 2.3 Dane a semantyczna koncepcja informacji.

Jednym z kluczowych pojęć dla całego uniwersum technologii komputerowych są „dane". Do tej pory rozważaliśmy jedynie teorie i koncepcje związane z zagadnieniem o nieco innym poziomie ogólności – informacją. W artykule *Semantic Conceptions of Information* Luciano Floridi prezentuje koncepcję definiowania pojęcia „informacji" w odniesieniu do danych właśnie (Floridi, 2013). Przy okazji zaproponowane zostaje umiejscowienie problemów związanych z semantyką w stosunku do ogólnej tematyki teorioinformacyjnej.



Ilustracja 1: rodzaje danych jako podstawy informacji.

Koncepcja włoskiego badacza polega na zastosowaniu tak zwanej Ogólnej Definicji Informacji (General Definition of Information, GDI) w celu sformułowania dającego się skutecznie operacjonalizować sposobu ujęcia zagadnień, które ze względu na swój silnie pierwotny charakter i wzajemne uwarunkowania są bardzo trudne do jednoznacznego zdefiniowania wprost. GDI wg. Floridiego daje się sformułować w sposób następujący:

„σ jest instancją informacji, rozumianej jako zawartość semantyczna wtedy i tylko wtedy gdy:

(GDI.1) σ zawiera jedną lub więcej daną

(GDI.2) dane w σ są dobrze uformowane[3]

(GDI.3) dobrze uformowane dane w σ mają znaczenie" (Floridi, 2013)

[3] W oryginale „*well-formed"* należy rozumieć jako posiadanie ustabilizowanej struktury, schematu, bycie zgodnym z pewnym zbiorem reguł dotyczącym kodowania, poprawność składniową.

Jak to zostało więc zaznaczone na początku GDI czyni użytek z kluczowego pojęcia *danych* w bardzo konkretnym ujęciu, bliskim technologiom informatycznym. Dalej, autor proponuje doprecyzowanie tej konstrukcji przez aplikację tzw. Diaforycznej Definicji Danych[4] (*Diaphoric Definition of Data, DDD).* Stanowi ona zwrócenie uwagi na fundamentalny fakt, że dla fizycznej (metafizycznej również) możliwości zaistnienia jakiegoś faktu konieczne jest pojawienie się jakiejkolwiek różnicy. Aby więc mówić o pewnym stanie, musimy mieć kryteria wyróżnienia go. Na poziomie koncepcyjnym charakter owej różnicy jest obojętny: może polegać ona zarówno na pojawieniu się czarnej plamy na białej kartce, zmianie ciśnienia powietrza (dźwięk), jak i na wahaniach stanu impulsu elektrycznego w układzie elektronicznym. Prowadzi to do fascynującego stwierdzenia o charakterze filozoficznym, że źródłem wszelkiej wiedzy, wszelkiego języka i całej komunikacji jest pewna fundamentalna niejednolitość uniwersum, a w szczególności naszej jego percepcji. Floridi pisze:

„W zależności od inklinacji filozoficznych DDD może być aplikowana na trzech poziomach:

1. Dane jako rozróżnienie *de re,* to jest wskazanie na pewien brak jednorodności w świecie zewnętrznym. Nie ma konkretnej nazwy na tego rodzaju »dane na wolności«. Możliwym rozwiązaniem jest określanie ich jako *dedomena* („dane" w języku greckim; zwróćmy uwagę, że nasze (czyli angielskie – dop. J.K.A.) słowo „data" („dane" w języku angielskim – dop. J.K.A.) pochodzi od łacińskiego tłumaczenia pracy Euklidesa pt. „Dedomena"). Dedomena nie powinny być mylone z danymi ze środowiska. Są one czystymi, proto- epistemicznymi danymi, to znaczy – danymi przed ich epistemiczną interpretacją. Jako »zaburzenia w tkaninie istnienia« mogą być jedynie postulowane jako zewnętrzne zahaczenie dla naszych informacji ponieważ dedomena nie mogą być nigdy dostępne i wyjaśniane niezależnie od pewnego poziomu abstrakcji. Mogą być rekonstruowane jako wymagane ontologicznie, na kształt kantowskich *noumenów* lub locke'owskiej *substancji*: nie są one epistemicznie doświadczane lecz ich obecność jest empirycznie wnioskowana z (i wymagana przez) doświadczenia. Oczywiście, nie możemy podać przykładów, ale dedomena są jakimkolwiek brakiem jednolitości co może stanowić (dla tych, którzy jak my przyglądają się systemom informacyjnym) źródło danych np. czerwone światło na czarnym tle. (...)

[4] Nie należy łączyć pojęcia „diaforyczności" z „diaforą" jako figurą retoryczną. Chodzi tu o odwołanie się do greckiego *diaphora* oznaczającego przesunięcie czy **różnicę**.

2. Dane jako rozróżnienie *de signo*, to znaczy brak jednorodności pomiędzy co najmniej dwoma stanami fizycznymi (ich percepcją), takimi jak wyższy czy niższy poziom naładowania baterii, zmiany sygnału elektrycznego podczas transmisji rozmowy telefonicznej lub kropki i kreski alfabetu Morse'a.

3. Dane jako rozróżnienie *de dicto*, to znaczy brak jednorodności pomiędzy dwoma symbolami, np. literami A i B w alfabecie łacińskim"

(Floridi, 2013)

Powyższe wyjaśnienia prowadzą do wniosku, że dla zaistnienia danych konieczny jest pewien fizyczny nośnik, materialna implementacja – pomimo tego, że ich „wartość" informacyjna w sensie epistemicznym jest arbitralna i zależy od przyjętego kodowania oraz przyjętych kryteriów semantycznych. Bardzo ładnie obrazuje to angielska fraza „bit from it" – bit (informacja na bazie danych) pochodzi z „czegoś" – w sensie „czegoś realnie istniejącego" i nie może być traktowana jedynie jako postulowana abstrakcja. Nawet najbardziej skrajne solipsyzmy muszą wykazać istnienie jakiejś podstawy dla danych, a więc dla informacji, a więc dla wiedzy. Był to jeden z najbardziej palących problemów wszelkich sceptycznych, racjonalistycznych systemów filozoficznych.

Wracając jednak do kwestii związanych z semantycznymi koncepcjami informacji – jak to zostało zaprezentowane, zaistnienie fenomenów o charakterze informacyjnym jest ściśle zależne od pojawienia się syntaktycznie dobrze uformowanych danych, zaistnienie tychże zaś bezpośrednio zależy od fizycznych możliwości implementacyjnych (ale: nie od *konkretnych* implementacji). Stąd medium - z punktu widzenia samego faktu zaistnienia informacji - nie ma znaczenia – podobnie jak system kodowania: język. Odgrywają one zaś kluczową rolę kiedy przychodzi do konkretnych procedur związanych z implementacją, przetwarzaniem i dystrybucją danych i informacji. Stąd twierdzenie, że rewolucja cyfrowa – jak to będziemy potem wykazywać – nie czyniąc rewolucji w samej charakterystyce naszego rozumienia rzeczywistości (nie będąc dla nas czymś ontologicznie nowym) może jednak spowodować całkowitą i nie do przecenienia zmianę w naszych sposobach działania w świecie.

Warto jeszcze w tym miejscu przytoczyć wyliczane przez Floridiego właściwości danych (zwane ich „neutralnościami"), ze względu na ich wysoką wartość porządkującą i wyjaśniającą pojęcie. Zdaniem filozofa dane charakteryzują się:

- neutralnością *taksonomiczną* tj. faktem, że nie ma z punktu widzenia GDI znaczenia identyfikacja danych (*data)* z konkretnymi odniesieniami (*relata)* w sensie fizycznym. W tym kontekście można powiedzieć, że „data are relata" („dane są relacjami") ale nie, że są „tymi tylko konkretnymi rodzajami relacji".

- neutralnością *typologiczną* tj. faktem, że informacje mogą zawierać wiele typów danych (pierwotne, drugiego rzędu, metadane, dane operacyjne, dane derywacyjne etc.)

- neutralnością *ontologiczną* tj. faktem, że informacja nie może istnieć bez reprezentacji danych; nie może być czegoś takiego jak informacja bez danych – co z kolei prowadzi do wniosku, że jest ona zależna od istnienia obiektów, w których można ją „zawrzeć". Nie mniej jednak dane dają się „przenosić" między swoimi implementacjami (np. prąd w kablu telefonicznym zamieniamy na drgania powietrza: dźwięk) co czyni je niesłychanie, pod tym względem, elastycznymi i stanowi źródło całego informacyjnego i informatycznego bogactwa świata komunikacji.

- neutralnością *genetyczną* tj. faktem, że dane jako relacje mogą mieć znaczenie (semantykę) niezależne od odbiorcy. Możemy z łatwością wyobrazić sobie przypadki gdy wiemy o czymś, że jest informacją (zawiera dane) ale nie rozumiemy jej. Istnieją teorie dopuszczające możliwość funkcjonowania danych posiadających znaczenie bez inteligentnego nadawcy/twórcy – nazywamy je m.in. *informacjami środowiskowymi* (vide: ilustracja), nie będą one jednak przedmiotem naszego szerszego zainteresowania.

Pozostaje jeszcze doprecyzować kwestie związane z GDI w kontekście przenoszenia znaczeń w komunikacji. Rzecz nie jest trywialna i pozostaje wiele otwartych problemów odnośnie tego jak z faktu fundamentalnej niejednorodności fizycznego uniwersum ludzie generują złożone konstrukcje poznawcze. Najbliższą teorioinformacyjnemu ujęciu wydaje się być koncepcja epistemologii znaturalizowanej Willarda Quine'a, która czyniąc użytek z koncepcji diaforycznej pozwala na prześledzenie aktywności poznawczych jako czynności związanych z rejestrowaniem bodźców i

tworzenia na ich podstawie umysłowych reprezentacji (Quine, 2006). Nie wdając się w szczegóły konkurujących teorii należy pamiętać o tym, że klucz do tego zagadnienia leży – znów – w kategorii „rozróżnienia". Dystynkcja rozumiana jako dający się wskazać stan, dana, jest wystarczającą podstawę do tego aby uczynić fakt znaczącym, przynajmniej potencjalnie. Na nasze tymczasowe potrzeby wystarczy techniczne rozumienie semantyki – takie jakie jest aplikowane w matematycznej teorii informacji oraz używane w konstrukcji Maszyny Turinga. Prawdziwe problemy związane z denotowaniem obiektów faktualnej rzeczywistości pojawi się na wyższym poziomie abstrakcji – kiedy zaczniemy zastanawiać się nad wyborem języka, za pomocą którego będziemy mogli opisywać świat, działać w nim a jednocześnie czynić go „zrozumiałym" dla komputera.

Σ 2.4 W stronę komputacji. Epistemologia jako teoria informacji 2.

Naszym wyjściem ze świata teorii informacji, która bardzo silnie redukuje pojęcie znaczenia, do stopnia w którym przestaje ono być zgodne z naszymi intuicjami na ten temat będzie ucieczka w system języka, który być może pozwoli nam na stworzenie interfejsu między techniczną teorią a codzienną praktyką semantyczną. Warto w tym miejscu wspomnieć jednak jeszcze jedną koncepcję, która stanowi próbę rozwiązania problemu epistemologii jako teorii informacji bez opuszczania uniwersum Shannona i Turinga. Chodzi o pomysł Gregory'ego Chaitina.

Koncepcja argentyńsko-amerykańskiego matematyka i informatyka stanowi nowe podejście do matematycznej teorii informacji w kontekście filozoficznym i bierze za punkt wyjścia teorie algorytmiczne oraz znane nam już twierdzenie Gödla o niezupełności. Epistemologiczność niezupełności i niedowodliwości systemów aksjomatycznych jest tu traktowana jako problem epistemologiczny właśnie, ze względu na ścisłą relację z kwestiami możliwości wskazania granic poznania matematycznego. Z naszego punktu widzenia najistotniejsze jest zaangażowanie w te rozważania Maszyny Turinga (jako mechanizmu „dowodzącego" koncepcji Gödla) oraz pojęcia danych i programu (opartego na tychże). Chaitin zaproponował sformułowanie liczby Ω (stała Chaitina) – liczby rzeczywistej, która reprezentuje prawdopodobieństwo, że losowo stworzony program (komputerowy, czyli Maszyny Turinga) zatrzyma się (halt). Pytanie, które interesuje badacza to: ile bitów informacji potrzebnych jest do zapisania programu? Inaczej: jaki jest najmniejszy program (w sensie danych, tak jak je definiowaliśmy powyżej) pozwalający obliczyć dany obiekt? Pisze on:

„Załóżmy, że na całym świecie istnieją tylko dwa programy, które kiedyś zatrzymają się oraz że te programy, kiedy przetłumaczymy je na ciągi bitów mają postać 11001 oraz 101. Wybranie jednego z nich losowo jest tym samym co losowe wygenerowanie tych ciągów. Możemy zrobić to rzucając monetą i zapisując 1 w przypadku „orła" i 0 w przypadku „reszki", tak że prawdopodobieństwo poszczególnego bitu wynosi $\frac{1}{2}$. Oznacza to, że prawdopodobieństwo otrzymania 11001 wynosi $\frac{1}{2}x\frac{1}{2}x\frac{1}{2}x\frac{1}{2}x\frac{1}{2}=\frac{1}{2}^5$. Więc prawdopodobieństwo losowego wybrania jednego z tych dwóch programów wynosi $\frac{1}{2}^3+\frac{1}{2}^5=0{,}15625$.

Oczywiście w rzeczywistości istnieje o wiele więcej programów, które się zatrzymają i Omega jest sumą wielu wyrażeń o postaci $\frac{1}{2}^N$. Ponadto kiedy definiujemy Omegę musimy poczynić pewne obwarowania co do tego, które programy są poprawne aby uniknąć obliczania rzeczy po dwa razy i by mieć pewność, że Omega nie stanie się nieskończenie duża.

Tak czy inaczej, przygotowując rzeczy poprawnie można zdefiniować prawdopodobieństwo zatrzymania Omega między 0 a 1. Omega jest porządną liczbą, która da się zdefiniować matematycznie."

(Chaitin, 2005b, p. 33)

Kiedy weźmiemy pewne wyrażenie matematyczne (Chaitin przytacza przykład √2 czyli liczby 1.14142135623730950488...) to możemy podać algorytm, który pozwala na konstruowanie rozwinięcia tego wyrażenia, cyfra po cyfrze aż do uzyskania zadowalającego nas stopnia dokładności (dla podanego przykładu może to być iteracja Newtona w postaci: $x_{k+1}=\frac{1}{2}(x_k+2/x_{k)}$ co dla $x_1=1$ daje 1,5, dla $x_2=1{,}5$ daje 1,41666... itd.). Widzimy więc, że skończony program może nam dać nieskończenie długie rozwinięcie pierwiastka z dwóch.

„Czy istnieje zatem skończony program, który może obliczyć wszystkie bity w rozwinięciu binarnym Omegi? Okazuje się, że wiedza na temat pierwszych N bitów Omegi daje nam możliwość rozwiązania problemu stopu dla wszystkich programów o N bitów długości. Gdybyśmy więc mieli skończony program, który pozwoli na wypracowanie wszystkich bitów Omegi, mielibyśmy skończony program, który pozwoli na rozwiązanie problemu stopu dla wszystkich programów, niezależnie od długości. Z tego co wiemy to niemożliwe. Taki program nie może istnieć."

(Ibid.)

Omega jest więc nieredukowalna, nie może być skompresowana do mniejszej, skończonej teorii. Pomimo tego, że Omega jest bardzo dobrze zdefiniowana – jest też maksymalnie niepoznawalna. Hipoteza Ω ma dla naszych rozważań na temat epistemologii jako teorii informacji niezwykłe konsekwencje:

„Centralny dla algorytmicznej teorii informacji model epistemologiczny stwierdza, że naukowa i matematyczna teoria jest programem komputerowym służącym do obliczania faktów, jak również to, że im ten program jest mniejszy, tym jest lepszy. (...) Skuteczność mocy wyjaśniającej danej teorii zależy od stopnia, w którym dokonuje ona kompresji liczby bitów dotyczących faktów na mniejszą liczbę bitów dotyczących teorii. Rozumienie jest kompresją!"

(Chaitin, 2005a, p. 33)

Σ 2.Apendyks: Jakiego kodu użyć?

W poprzednich częściach pracy wiele miejsca poświęcone zostało prezentacjom aplikacji kodu binarnego. Głównym powodem stosowania systemu zerojedynkowego do rozważań teoretycznych jest jego prostota (wiemy, że aby mieć dane musi zajść pewna *różnica* – np. taka między 0 a 1). Podstawową prawdą na temat bitów jest to, że każdą informację, która da się sprowadzić do wyboru między dwoma czynnikami da się zapisać za ich pomocą. Lecz co z bardziej złożonymi kwestiami? Tu również system binarny wykazuje swoją poręczność, oto bowiem każdy dodatkowy bit podwaja liczbę kodów (kombinacji), które można za pomocą zestawu zer i jedynek zapisać: 1 bit daje nam 2^1 możliwych kombinacji czyli dwie możliwości; 2 bity dają ich 2^2 czyli 4, 3 bity 2^3 to jest 16; zaś jedynie 10 bitów to aż 2^{10} = 1024 kombinacje czyli jeden kilobit potencjalnej informacji. 8 bitów (b) daje 1 bajt (B). 1024 B czyli jeden kilobajt to 2^{10} x 8 bitów. Wczesne komputery osobiste miały standardowo dyski pojemności liczonej w dziesiątkach i setkach megabajtów (MB czyli 10^6B), dziś standard dla komputera domowego to wielokrotności terabajtów (TB, 10^{12}B). Jako ciekawostkę można podać, że CERN'owski LHC czyli Wielki Zderzacz Hadronów wygenerował w ciągu około roku pracy i testów mniej więcej 100 petabajtów (PB) danych, co oznacza 8 x 100 000 000 000 000 000 stanów binarnych. To jeśli chodzi o aspekt ilościowy w kwestii współczesnych możliwości zapisu cyfr binarnych.

Drugim istotnym faktem, który stanowi źródło powszechności takiego a nie innego systemu kodowania jest, oczywiście, logika. Nie tylko ta matematyczna ale przede wszystkim – logika przetwarzania danych. Historia rachunku binarnego zaczyna się mniej więcej w XVII wieku za sprawą przede wszystkim wspominanego już niejednokrotnie Gottfrieda Wilhelma Leibniza, który od ojca Joachima Bouveta – wysłannika do Chin z ramienia dworu Ludwika XIV otrzymał 64 heksagramy księgi „I Cing". Zainspirowały one filozofa do przepisania ich na jedynki (linia ciągła) i zera (przerwa). Zauważył, że heksagramy dają się przepisywać na kolejne liczby naturalne:

Ilustracja 2: Hexagramy Księgi Przemian (Yijing)

A = 000000 = 0

B = 000001 = 1

C = 000010 = 2 itd.

Leibniz zachwycony prostotą i elegancją tego rodzaju zapisu uznał, że może on stanowić podstawę języka uniwersalnego. Co interesujące, o ile kombinatoryka rzeczywiście miała w chińskich systemach myślenia swoje ważne miejsce o tyle binarna interpretacja „Księgi Przemian" przez europejskiego matematyka była zdecydowanie zbyt daleko posunięta, jako że w Państwie Środka nie znano liczby „0" do bardzo późnego momentu (Ifrah, 2006, p. 732). Tak czy inaczej w 1679 roku Leibniz napisał pracę *De progressio dyadica*, gdzie wyraził swoją fascynację systemem dwójkowym. Nie był to z całą pewnością pierwszy w historii myśli europejskiej tekst rozważający możliwość takiego rachunku lecz z całą pewnością wywarł on duży wpływ (wraz z *Rozważaniami o dwójkowej arytmetyce* z 1703 roku, które spowodowały swoistą modę na „binarność" w Towarzystwie Królewskim i wśród sympatyków) na ówczesnych matematyków. Rozważania Leibniza, choć zorientowane matematycznie i praktycznie (planował on skonstruowanie urządzenia liczącego opartego na systemie dwójkowym, powstał nawet prototyp) miały jednak silny rys mistyczny i teologiczny. Kwestie tego właśnie rodzaju odwróciły ostatecznie uwagę filozofa od prób skonstruowania pełnych reguł binarnego systemu logicznego.

Przeszło sto lat później temat tego rachunku wrócił w pismach George'a Boole'a, który opracował rodzaj algebry (nazwanej od jego imienia). Struktura ta składa się z:

działań dwuargumentowych:

∨ - alternatywy (lub; OR),

∧ - koniunkcji (i; AND),

operacji jednoargumentowej:

¬ - negacji (nie, NOT) ,

oraz elementów:

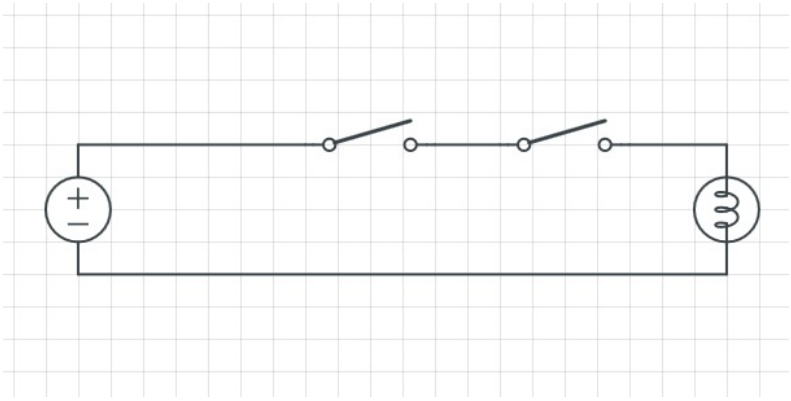
0 (fałsz; FALSE) i 1 (prawda; TRUE).

Operacje algebry Boole'a mogą być wykonywane na słowach bitowych tak, że biorą w nich udział odpowiednie bity na odpowiednich pozycjach w zestawianych słowach np.

negacja: $\frac{1100}{0011}$; alternatywa: $\frac{1010}{1100}{1110}$; koniunkcja: $\frac{1010}{1100}{1000}$ etc.

W ramach algebry Boole'a obowiązują prawa: łączności, przemienności, absorpcji, rozdzielności, odwrotności oraz prawa w stosunku do nich pochodne a także prawa De Morgana.

Istnienie algebry Boole'a i systemu binarnego byłyby jedynie ciekawostkami matematycznymi gdyby nie fakt, że dają się one wyśmienicie implementować w układach elektrycznych. Wyobraźmy sobie następującą sytuację: mamy do dyspozycji żarówkę i baterię oraz łączące je kable i dwa włączniki (otwierające i zamykające obwód).



Ilusracja 3: Prosty elektryczny układ szeregowy.

Jeśli połączymy cały układ szeregowo to uzyskamy następujące możliwe stany układu:

Wyłącznik	Wyłącznik	Żarówka
Wyłączony	Wyłączony	Nie świeci
Wyłączony	Włączony	Nie świeci
Wyłączony	Wyłączony	Nie świeci
Wyłączony	Włączony	Świeci

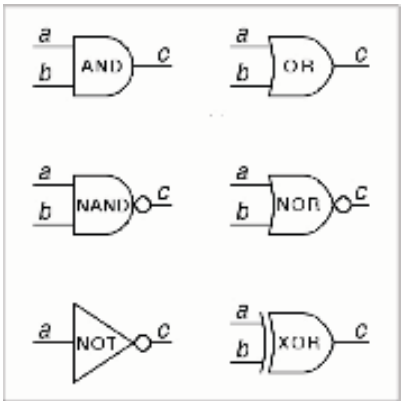
Przyjmijmy, że stany włącznika odpowiadają wartościom binarnym, zaś żarówka wynikowi działania dwuargumentowego:

α	β	$\alpha \wedge \beta$
0	0	0
0	1	0
1	0	0
1	1	1

Jak widzimy na układzie elektrycznym możemy w bardzo prosty sposób zasymulować operację koniunkcji dwóch stanów (AND). W podobny sposób – układając włączniki równolegle – uzyskamy symulację operacji alternatywy (OR). Obwody możemy oczywiście łączyć sieci i tworzyć w ten sposób modele bardziej złożonych operacji logicznych.

W ten oto sposób nasze rozważania zataczają pętle ponieważ zastosowanie obwodów elektrycznych do reprezentacji działań logicznych było jednym ze wczesnych konceptów Claude Shannona, który na ten temat napisał pracę magisterską na M.I.T. w 1938 roku *A Symbolic Analysis of Relay and Switching Circuits*, jego tekst wprowadzający znane nam już pojęcie bitu został napisany dziesięć lat później.

Prezentowany przykład stanowi oczywiście ogromne uproszczenie, jego rozwinięciem może być zastosowanie „gotowych” bramek NOT, AND, OR a także NOR (notOR) i NAND (notAND) i w końcu XOR (exclusiveOR), które powodują, że można uniknąć skomplikowanego kreślenia układów składających się z wielu warstw.



Ilustracja: Podstawowe bramki logiczne

Dysponując koncepcją kodu dwójkowego, systemem logicznym umożliwiającym wykonywanie na nim operacji logicznych przełożonym przez Shannona na świat elektroniki oraz – później – zyskując narzędzie algorytmiczne w postaci Maszyny Turinga znaleźliśmy się u progu możliwości pojawienia się pierwszego, współczesnego komputera. Zanim jednak przyjrzymy się mechanizmom działania współczesnych maszyn obliczeniowych i – co ważniejsze – działającego na nich oprogramowania musimy zastanowić się nad drugą stroną informatycznego „medalu" czyli językiem i jego relacjom w stosunku do omawianych zagadnień.

Σύστημα 3: Komputer i język.
Od alfabetu do algorytmu.

„4.121
Zdanie nie może przedstawiać formy logicznej; ona się w nim odzwierciedla.
Co się w języku odzwierciedla, tego język nie może przedstawiać.
Co się w języku samo wyraża, tego my wyrazić przezeń nie możemy.
Zdanie pokazuje logiczną formę rzeczywistości.
Ono ją przejawia.”

(Wittgenstein, 2000, p. 28)

„Język jest »usystematyzowaniem« ekspresji. Symbole są wyspami względnej pewności w morzu znaczeń – choć zdeterminowane w kształcie i swoim rozwinięciu, ich »linie brzegowe« są nieskończone (by przywołać obraz z Teorii Chaosu). Są twardą walutą myśli. (...) Systemy symboli tworzą kody (...) Istnieje dobry powód aby rozróżniać między prawdziwymi kodami czy językami – systemami znaków stojącymi za konceptualnymi rzeczywistościami innymi niż one same, gdzie zarówno znaki jak i te rzeczywistości są dostępne dla użytkownika, a innymi użyciami słów »kod« czy »język« w których taka relacja nie zachodzi. W związku z tym »kod genetyczny« (gdzie wiązania DNA nie oznaczają żadnej zewnętrznej rzeczywistości lecz odnoszą się raczej do swoistej instrukcji jej konstruowania) ani żaden rzekomo uniwersalny »kod mózgowy« niedostępny dla swoich użytkowników nie podpadają pod tą definicję”

(Fortescue, 2001, pp. 53-54)

„Komputer traktuje język tak samo jak traktuje logikę, przestrzeń i czas, osobliwie łącząc praktycyzm z abstrakcją filozoficzną. Na poziomie praktycznym języki komputerowe są kodami, których celem jest odzwierciedlenie struktury logicznej problemu. Są one oddalone, jak to tylko możliwe, od emocjonalnego, wieloznacznego i wokalizowanego języka życia codziennego. (...) Gdy specjalista komputerowy tworzy swoje kody, bierze tym samym udział w debacie, która toczy się z ożywieniem od tysięcy lat. Czy język powinien ewokować czy denotować? Czy wchodzi on w magiczny, tajemniczy związek z myślami i przeżyciami, które opisuje, czy też jest przede wszystkim instrumentem ujawniania rzeczywistej struktury świata? Czy język jest zasadniczo poetycki, czy logiczny?”

(Bolter, 1990, p. 190)

Σ 3.1 Źródła. W Poszukiwaniu języka uniwersalnego.

Język jest chyba najbardziej fascynującym fenomenem i największą zagadką ludzkiego świata. Umberto Eco w książce *Poszukiwanie języka doskonałego* podkreśla ogrom różnorodności tematów i treści jakie napotka badacz chcący zmierzyć się z tematem funkcjonowania języka w samych tylko ramach kultury europejskiej i to uwzględniając praktycznie jedynie te sztucznie konstruowane:

„Borst na omówienie samych tylko dyskusji wokół pomieszania języków potrzebował aż sześciu woluminów, gdy zaś skończyłem niniejsze wprowadzenie pojawiło się dzieło Demonet, która samym tylko rozważaniom nad naturą i genezą języka w latach 1480-1580 poświęciła bez mała 700 zadrukowanych stronic. Couturat i Leau analizują dość gruntownie 19 modeli języków apriorycznych i 50 języków mieszanych i aposteriorycznych; Monnerot-Dumaine rejestruje 360 projektów języków międzynarodowych; Knowlson z kolei wylicza 83 prace poświęcone wyłącznie modelom języków uniwersalnych powstałym w XVII i XVIII stuleciu, wreszcie ograniczający się do projektów dziewiętnastowiecznych Porset dorzuca 173 tytuły”

(Eco, 2013, p. 15)

Wobec takiej różnorodności łatwo się zagubić. Eco na potrzeby swoich rozważań dokonał uproszczonej klasyfikacji tematów (dzieląc je również na te, którymi chciałby się zająć i na te, które musi pominąć):

I. Ponownie odkryte języki historyczne, uznawane za pierwotne i nacechowane mistyką doskonałości (hebrajski, egipski, chiński),
II. Rekonstrukcje języków uważanych za pierwotne (laboratoryjne prajęzyki takie jak na przykład indoeuropejski),
III. **Języki budowane sztucznie**, które miały służyć doskonałości funkcjonalno-strukturalnej, uniwersalności lub też konkretnej praktyce (kryptografia),
IV. Języki „magiczne” i mistyczne.

A. Języki oniryczne (język chorych psychicznie, transowy, mistyczny),
B. Języki literackie (takie jak nowomowa Orwella czy języki tolkienowskiego Śródziemia),
C. Języki bricolage (mieszane języki na granicach spotkań cywilizacji),

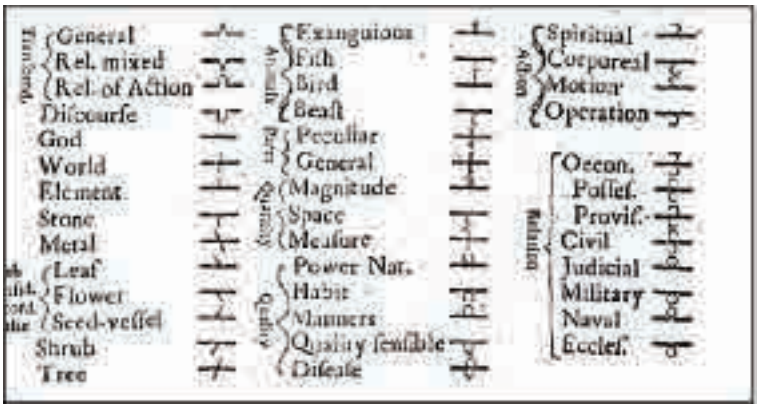
D. Języki pośredniczące (żargony na terenach wielojęzycznych),
E. Języki specjalistyczne i formalne,
F. „Szaleństwa językowe”.

Ponieważ jednak na kartach niniejszej dysertacji planujemy przyjrzeć się kwestiom związanym z działaniem oprogramowania, które w swojej istocie jest przecież wykonywanym przez komputery zbiorem poleceń, konstrukcją ze wszech miar językową (choć specyficzną) nie możemy uniknąć ogólniejszej refleksji na temat języka. Inwestygacja włoskiego semiotyka tropem europejskich języków uniwersalnych wydaje się dobrym punktem wyjścia do takich rozważań. Jak się bowiem okaże języki programowania, komputer i technologie informacyjne są pokłosiem marzeń o zrozumieniu działania *mechaniki* procesów lingwistycznych w równym stopniu co prezentowanych prób uchwycenia *prawdy* w procesach poznawczych. Zaczniemy jednak nie – jak to czyni Eco – od Adama i Ewy oraz biblijnej *Księgi Rodzaju*, lecz znacznie bliższego nam okresu, kiedy Rozum i System były podstawowymi motywacjami dla europejskich uczonych.

Σ 3.1.1 Język filozoficzny Wilkinsa i *lingua characteristica* Leibniza.

John Wilkins (1616-1672) biskup Chester to postać niezwykle barwna, nawet na tle swojej niebanalnej epoki. Współzałożyciel angielskiego Towarzystwa Królewskiego (The Royal Society of London for Improving Natural Knowledge) – organizacji, wokół której oscylowali najważniejsi „filozofowie naturalni” tamtych czasów (należał do nich Izaak Newton, bywał gościem Gottlieb Leibniz), a także niezliczona ilość mniej lub bardziej interesujących osobowości zainteresowanych uprawianiem baconowskiej „nowej nauki”. Autor z jednej strony fantastycznego traktatu *The Discovery of a World in the Moone*, z drugiej zaś zawzięty empirysta i głosiciel prawdy opartej na rozumie. Jego największą – z naszego punktu widzenia – zasługą było opracowanie bardzo specjalnego, sztucznego języka uniwersalnego zaprezentowanego na kartach *An Essay towards a Real Character and a Philosophical Language* (1668). Rzecz polegać miała na stworzeniu struktury o charakterze lingwistycznym, która nie popadnie w retoryczne formuły, pozwalając na jedno-jednoznaczny zapis relacji język – rzeczywistość. Bez możliwości popadnięcia w poetyczne abstrakcje i tautologiczne (w sensie – nic nie wnoszące) wypowiedzi. Język Wilkinsa miał być, jakbyśmy to dzisiaj powiedzieli, bezpośrednim i całkowicie przeźroczystym interfejsem między światem umysłu a uniwersum empirycznych obiektów. Struktura języka miała za zadanie odpowiadać strukturze rzeczywistości. Środkiem do

tego celu miało być stworzenie potężnych diagramów obejmujących taksonomię całej Natury. Filozofowi, na wstępnym etapie prac udało się stworzyć tabelę 40 rodzajów z 251 zróżnicowaniami, co daje podział na 2030 gatunków. Do tabeli dodana zostaje gramatyka filozoficzna, w ramach której wprowadzone zostają oznaczenia wyrazów pochodnych i system morfemów – tak, że możliwa staje się transkrypcja (przynajmniej w pewnym zakresie) języka naturalnego na nowo powstały twór. Następnie naturalista zaproponował już właściwy język, którego część pisana polegać miała na użyciu ideogramów oznaczających – w uproszczeniu – gatunki podstawowe i różnorakie relacje.



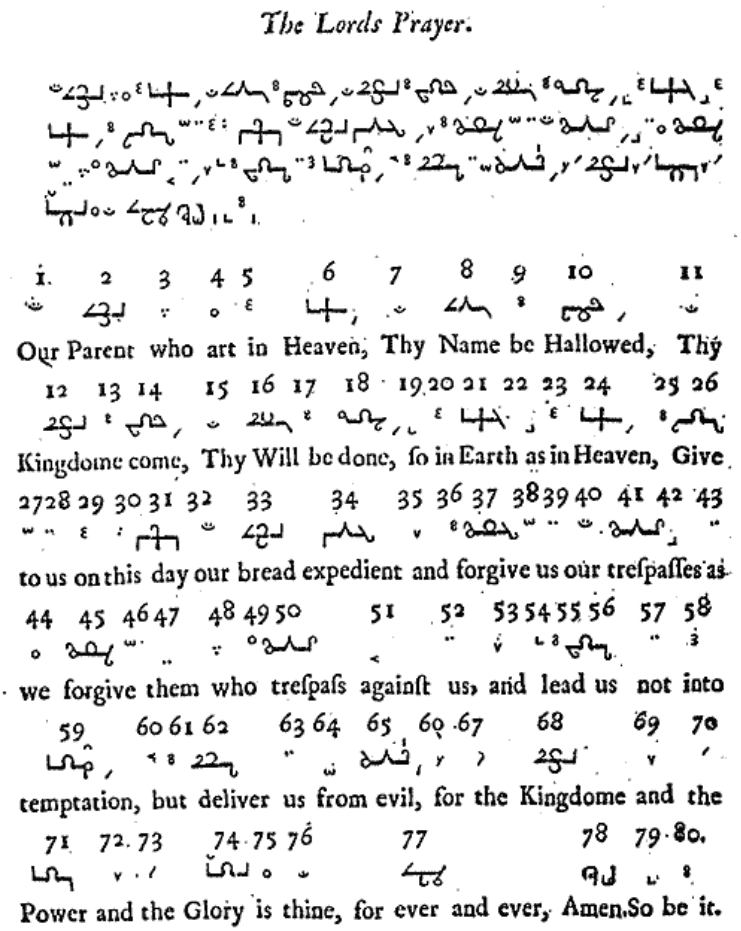
Ilustracja 5: *Pierwsza warstwa* klasyfikacji Wilkinsa

Do ideogramów dołączona została tabela obrazująca sposoby wymowy poszczególnych znaków.

„Wilkins podaje tu następujący przykład:

Jeśli (De) znaczy element to (Deb) musi znaczyć pierwszą odmianę, którą (według Tabel) jest Ogień, a (Debá) znaczyć będzie pierwszy Gatunek to jest Płomień. (Det) będzie piątą różnicą w tym Rodzaju, który jest Pojawiającym się Meteorem, (Debá) – pierwszym jego Gatunkiem to jest Tęczą; (Deta) drugim, czyli Aureolą.

A oto zapis pierwszych słów modlitwy *Ojcze nasz*[5]:



Ilustracja 6: modlitwa *Ojcze Nasz* w języku Wilkinsa (źródło: Wilkins, 1666).

Znak pierwszy wskazuje zaimek dzierżawczy w pierwszej osobie l.mn, drugi – Stosunki Ekonomiczne, kreseczka ukośna po lewej stronie odnosi się do pierwszej Różnicy (więzi pokrewieństwa), kreseczka ukośna po prawej stronie – do drugiego Gatunku, Bezpośredniego Przodka w linii wstępnej. Dwa pierwsze znaki należy zatem czytać jako nasz ojciec a czytać jako *hai coba*"

(Eco, 2013, p. 266)

Z praktycznego punktu widzenia, na ówczesne czasy projekt Johna Wilkinsa nie mógł zostać bez gigantycznego nakładu sił i środków zrealizowany. Fiasko projektu biskupa Chester jako rzeczywistego narzędzia poznawczego powoduje jednak zwrócenie naszej uwagi na inny jego aspekt. Jak zauważa Eco, filozof dążył do skonstruowania czegoś co dzisiaj nazwalibyśmy strukturą hipertekstową. Spostrzeżenie to możemy poszerzyć o obserwację dotyczącą bazodanowego charakteru całego przedsięwzięcia.

[5] Ilustracja nie pochodzi z cytowanego fragmentu książki lecz ze skanu oryginalnego dzieła.

Różnica między hipertekstem, a bazą danych polega przede wszystkim na tym, że ten pierwszy jest całkowicie arbitralny w swoich połączeniach natomiast drugi przypadek stanowi rodzaj organizacji danych podlegający pewnemu konkretnemu modelowi (choć nie musi on w żaden sposób odpowiadać rzeczywistości i w tym sensie jest też arbitralny, choć na innym poziomie) tak aby wspomagać proces pozyskiwania informacji. Drzewo klasyfikacyjne Wilkinsa charakteryzuje się pewną spójnością pod względem logicznym i posiada dobrze zapostulowany (choć niezbyt skutecznie realizowany) model. Istnieją specyficzne języki, które umożliwiają pracę ze współczesnymi bazami danych na poziomie definicji typów danych i relacji między nimi, zarządzania zawartością (dodawania sensownych wpisów, aktualizacji) oraz – co najważniejsze – wyszukiwania i dostarczania informacji (w tym sumowania, uśredniania, sortowania, grupowania, krzyżowania treści). System Wilkinsa moglibyśmy nazwać proto- silnikiem *(database/storage engine)* pewnej specyficznej bazy danych, brakuje w nim jedynie porządnych mechanizmów automatyzacji przetwarzania informacji – inaczej mówiąc w XVII wiecznym systemie (znów: gdyby język Wilkinsa miał nieco lepszą implementację) filozof naturalny musiałby, pomimo umiejętności posługiwania się specjalistycznym językiem, dochodzić do prawd samodzielnie wyłuskując je ze struktur i tabeli. Optymalnie byłoby, skoro dysponujemy przyzwoitym modelem, rzecz zautomatyzować. I tu znów wkracza w obszar naszych zainteresowań Leibniz.

W poprzednich rozdziałach nieco miejsca poświęcone zostało niektórym zainteresowaniom niemieckiego filozofa, szczególnie podkreślając jego zasługi dla rozwinięcia pozycyjnego systemu binarnego. Zagadnienie to łączy się bezpośrednio z pragnieniem Lebniza aby stworzyć język myśli, który nie tylko pozwoliłby na adekwatne mówienie o rzeczach ale przede wszystkim na przeprowadzanie niezawodnych rozumowań. Gottlieb Wilhelm zapoznał się z dziełem Wilkinsa około roku 1671 i pomimo niechęci do zastosowanej przez autora nowej transkrypcji, wyrażał się o nim z uznaniem. Problemem projektu powstającego pod auspicjami Towarzystwa Królewskiego była przede wszystkim jego nadmierna deskryptywność, skupienie na opisie zamiast na wyrażaniu logicznych relacji między pojęciami, co miało swoje źródło w silnie empirycznej orientacji członków stowarzyszenia. Marzeniem Leibniza natomiast było stworzenie rachunku – algebry myśli. Pomysły swoje zawarł najpierw w koncepcji Języka Ogólnego – rzecz polegała na próbie przyporządkowania pojęciom wartości liczb naturalnych tak aby działania na nich przeprowadzane mogły być poprawnymi złożeniami np. jeśli liczba 2 to „zwierzę”, zaś 3 to „rozum” wtedy, jeśli 6 oznaczałoby „człowieka” to moglibyśmy zapisać równanie 2 x 3 = 6 czyli prawdę zarówno pojęciową jak i matematyczną. Sprawa

oczywiście nie była tak prosta jak w prezentowanym przykładzie, również na poziomie leksykalnym. Stwierdzenie, że język jest arbitralnym, warunkowanym kulturowo kodem i daje się nim manipulować tak jak to czynili Wilkins, Delgrano i inni stało się szybko stosunkowo łatwe do zaakceptowania. Znaczniej mniej oczywistą okazuje się kwestia odniesienia tej tezy do świata liczb.

„Jedyny sposób aby oczyścić nasze rozumowania to uczynić je tak wyrazistymi jak te Matematyków, tak abyśmy mogli odnajdywać nasze błędy natychmiast i kiedy zaistnieje spór między osobami, będziemy mogli powiedzieć: Obliczmy to [*calculemus*], bez zbędnych ceregieli, aby przekonać się kto ma rację”

(Leibniz, 1951)

Aby zrealizować swoje zamierzenie filozof rozwijał rozważania nad językiem w stronę projektu *characteristica universalis* mającego stanowić próbę symbolicznej reprezentacji pojęć matematyki, empirycznej fizyki i metafizyki razem. Znaki tego systemu miały bezpośrednio korespondować z ideami, a jednocześnie dawać się składać w zdaniopodobne syntaksy. Drugą częścią planu był *calculus ratiocinator* – mechanika przeprowadzania inferencji na wyrażeniach należących do *characterisicae*. Możemy o niej myśleć podobnie jak o konceptualnej maszynie opracowanej przez Turinga – tworze ściśle matematycznym, który jednak w odpowiednich okolicznościach dało się zamienić na konkretne urządzenia. Wiemy, że Leibniz pracował nad prototypem, który osiągnął swą postać jako „Rachmistrz Krokowy” albo „Żywa Ława Obliczeniowa”, mechanizm oparty na wynalezionym przez filozofa specjalnym systemie biegów. Powstało coś w rodzaju mechanicznego, napędzanego przekładnią kalkulatora – koncept nie całkiem nowy – podobne urządzenia konstruował m.in. Pascal. Z dzisiejszej perspektywy wiemy, że marzenie Leibniza nie mogło się spełnić w takim kształcie jaki sobie zaplanował ze względów ściśle matematycznych. Znamy losy XX wiecznej matematyki i zmagania Kurta Gödla oraz Alana Turinga, którzy w końcu wykazali, że ogólnie – dla żadnego odpowiednio złożonego systemu nie istnieje procedura algorytmiczna mogąca służyć do oddzielenia wyrażeń prawdziwych od fałszywych w skończonym czasie. Niemiecki filozof, tworzący swoje dzieła bez mała trzysta lat wcześniej nie mógł oczywiście być świadomy tego ograniczenia. Nie mniej jednak to dzięki takim pionierom jak Wilkins czy Leibniz otworzyła się droga do komputacji – która w swej działającej postaci pomimo, że nie jest ostatecznym spełnieniem nadziei pokładanych w *characteristica universalis* to jednak zmieniła na zawsze oblicze zachodniej cywilizacji. Sam pomysł realizacji algebry myśli i

operującego na niej urządzenia pozostał żywy bardzo długo, jego bodaj najbardziej znaczącymi próbami wprowadzenia były opisywany już „Begriffsschrift" Fregego, o którym autor pisał wprost, że pragnie stworzyć nie tylko *calculus racionitator* ale właściwą *lingua characteristica* w sensie leibnizowskim. Na poziomie technicznym najbardziej istotną kontynuacją prac myśliciela z Lipska była Maszyna Różnicowa lorda Babbage'a, która choć nie do końca zrealizowana (zabrakło technologicznej precyzji... i pieniędzy; w pełni działającą maszynę odtworzono dopiero na początku lat 90-tych XX wieku) jest wyśmienicie opisana teoretycznie, łącznie z pierwszymi w historii przykładami algorytmicznego oprogramowania autorstwa Ady Lovelace (inne urządzenia programowalne były znane już wcześniej w postaci pozytywek z wymiennymi melodiami czy krosien Jacquarde'a). Opracowania Charlesa Babbage'a (maszyna Mark II) posłużyły Alanowi Turingowi do stworzenia schematu pierwszego prawdziwego, we współczesnym sensie, komputera na kartach słynnej pracy *On Computable Numbers* (*O liczbach obliczalnych*).

Σ 3.2 Język i algorytm.

Marzenie zachodnich filozofów o stworzeniu języka uniwersalnego ziściło się w drugiej połowie XX wieku, choć w dość niespodziewany sposób. Jednak jeśli przyjrzymy się dokładnie postulowanym funkcjom takiego języka, który miałby służyć:

- jako język pomocniczy komunikacji międzynarodowej, pozwalający na komunikowanie się ludzi o różnych językach ojczystych;
- jako język nauki, dostarczający uproszczonego systemu symbolicznego na opis wiedzy naukowej;
- jako rachunek logiczny, dostarczający instrumentu poznania i dowodzenia wypowiedzi;

to możemy bez dużej przesady powiedzieć, że algorytmiczny język komputerowego oprogramowania spełnia poniekąd te wytyczne. Czyni to w specyficzny sposób, zdecydowanie nie tak jak wyobrażali to sobie Wilkins, Leibniz, Babbage i wszyscy inni spośród tych, którzy podejmowali refleksję nad automatyzacją języka i wiedzy w czasach przed XX wieczną rewolucją logiczno-informatyczną.

Jak więc działa język oprogramowania na poziomie ogólniejszym niż opisywane do tej pory kwestie związane z kodowaniem i rachunkiem logicznym? Czy w perspektywie

modelowej uwidacznia on inne rodzaje relacji niż historyczne lub ściśle matematyczne języki formalne?

Sztuczne języki logiczne służą wyrażaniu prawd w oparciu o pewną aksjomatykę oraz reguły inferencyjne. Prawda jest rozumiana jako relacja między twierdzeniem języka formalnego a pewnym modelem „świata". Derywaty twierdzeń (syntaksy), relacje między twierdzeniami a modelem (semantyka) oraz sam model są formułowane w postaci systemu, za pomocą metajęzyka. Niewątpliwą zaletą takiego rozwiązania jest gwarantowana i sprawdzalna spójność i jednoznaczność wnioskowania. Wadą, trudność przeniesienia wyników na płaszczyznę empiryczną czy komunikacyjną.

Języki naturalne nie powstały w wyniku kompletnej operacji konstruowania zamkniętego modelu, co oznacza, że charakteryzują się fundamentalną niejednoznacznością ale też i niebywałą elastycznością w opisie fenomenów (już nie „świata" lecz po prostu – świata). Chcąc przełożyć jakikolwiek język naturalny na dowolny język formalny (w szczególności – automatycznie przetwarzalny) napotkamy na szereg trudności z tym związanych. Przede wszystkim języki naturalne rozróżniają między znaczeniem bezpośrednim znaków i wielością poziomów referencji kontekstualnych, metaforycznych, które muszą zostać poznane przez zinternalizowanie konwencji ich użycia (w procesach poznawczych i aktywnościach, interakcjach) podczas gdy w przypadku języków formalnych za relację odnoszenia się odpowiada metajęzyk.

Języki komputerowe stanowią w tej klasyfikacji osobny przypadek. Z jednej strony, nie są one abstrakcjami - służą bowiem specyficznemu rodzajowi komunikacji w pewnym kontekście. Jej celem jest wykonanie pewnego proceduralnego zadania (*procedural execution*). System komunikacyjny, w którym elementem jest programowany komputer nie jest więc obiektem statycznym (tak jak system logiczny, poza ramy którego nie ma wyjścia). Z drugiej strony języki komputerowe oparte są na konkretnych modelach, z reguły będących uproszczonymi reprezentacjami problemów, których rozwiązaniu mają służyć. Relacje syntaktyczne są tu również ściśle zdefiniowane, wymagana jest pełna precyzja „wypowiedzi" (czyli kodu). Struktura języka komputerowego rozłożona jest między dwoma biegunami. Z jednej strony mamy bowiem kod maszynowy przeznaczony do bezpośredniego wykonania przez maszynę (procesor), język assemblera będący na tyle niskopoziomowy na ile to możliwe bez utraty zrozumiałości dla człowieka – z drugiej świat opisu słownego, potocznego, w którym chcielibyśmy mówić o rzeczach i tym co można z

nimi zrobić. Jak się okazuje, da się znaleźć złoty środek wyważający te dwie racje, o czym będziemy się upewniać w kolejnych częściach niniejszej dysertacji.

Wróćmy jednak do kwestii związanych z refleksją na temat języka w kontekście algorytmicznym. Przyjmijmy roboczą definicję, zgodnie z którą algorytm to:

„Skończony ciąg/sekwencja reguł, które aplikuje się na skończonej liczbie danych, pozwalający rozwiązywać zbliżone do siebie klasy problemów;
Zespół reguł charakterystycznych dla pewnych obliczeń lub czynności informatycznych;"

(Wróblewski, 2010, p. 18)

Samo słowo „algorytm" pochodzi od łacińskiego tłumaczenia imienia Abu Abdullah Muhammad ibn Musa Al-Khwarizmi'ego, perskiego matematyka, który żył w IX wieku i uchodzi za autora jednego z pierwszych traktatów na temat rachunków zmiennych (wyraz „al-Jabr" w tytule tekstu: *Al-Kitab al-Mukhtasar fi Hisab al-Jabr wa l-Muqabala* czyli *Zasad redukcji i przenoszenia* był źródłem wyrazu „algebra"). Jednym z najbardziej znanych przykładów wczesnego algorytmu jest opracowana przez Euklidesa metoda odnajdywania największego wspólnego dzielnika dwóch liczb naturalnych: wystarczy wziąć parę postaci – mniejsza z liczb oraz różnica większej z mniejszą; proces należy powtarzać aż uzyskamy parę równych liczb. Stanowią one największy wspólny dzielnik wyjściowej pary. Pseudokod (struktura przypominająca zapis w języku programowania ale nie będąca implementacją w żadnym konkretnym języku) takiego algorytmu wyglądałby następująco:

```
NWD (a,b)   { dopóki b ≠ 0
              c := reszta z a/b
              a := b
              b := c
              zwróć a}
```

Na powyższym przykładzie widać doskonale, że algorytmy nie są (czy też nie muszą być) programami. Stanowią po prostu zbiór procedur transformujących pewien zbiór elementów wejściowych w elementy wyjściowe. Procedury te mogą być zastosowane zarówno w cyfrowych urządzeniach do elektronicznego przetwarzania danych jak i w życiu codziennym (np. w postaci instrukcji, przepisu kulinarnego). Mają one swoje kluczowe miejsce w świecie komputerów, który bez nich nie mógłby działać –

stanowią bowiem opis instrukcji jakie należy opracować dla maszyny, aby mogła ona zrealizować operacje na danych zgodnie z intencjami autora kodu. Wiemy, że komputery potrafią wykonywać pewne serie działań na bitach, są bowiem przecież „wdrożeniami" Uniwersalnej Maszyny Turinga; wiemy też, że różnego rodzaju dane mogą być zapisywane i przetwarzane w postaci binarnej; algorytmy są tym ogniwem, które pozwala nadać całemu systemowi dynamikę: mówią o tym, które dane wziąć i co tak naprawdę z nimi zrobić. Ich realizacje w poszczególnych językach programowania to już kwestia bardziej techniczna niż merytoryczna.

Z poszerzonego o pojęcie algorytmu krajobrazu naszych rozważań wynurza się interesujące spostrzeżenie: być może wszystkie problemy z formalizacją języka, wynikające z jego kontekstualności oraz dynamicznego charakteru mają charakter właśnie algorytmiczny – czyli pragmatyczny. Zwraca na to uwagę Andrew Goffey w swoim tekście zawartym w zbiorze *Software Studies. A Lexicon* (Fuller, 2008):

„Postrzeganie algorytmów w ten sposób, jako wypowiedzi w ramach dyskursu maszynowego może problematyzować ich istnienie w ten sposób, że podważa dychotomie »czyste/stosowane« czy »teoria/praktyka« (...) Formalistyczne dążenie do domknięcia zawartości form abstrakcyjnych nie tylko opiera się na rozróżnieniu teoria/praktyka ale też wydaje się wyłączać analizę związku między kluczowymi bytami informatyki (computer science) a historycznym kontekstem. Tylko dlatego, że tworzenie algorytmów wymaga de facto pewnego poziomu formalnej abstrakcji, która pozwala algorytmom na bycie aplikowanymi do różnego rodzaju zawartości, nie oznacza, że wyczerpaliśmy wszystko co powinniśmy wiedzieć aby zrozumieć procesy, których są one częścią."

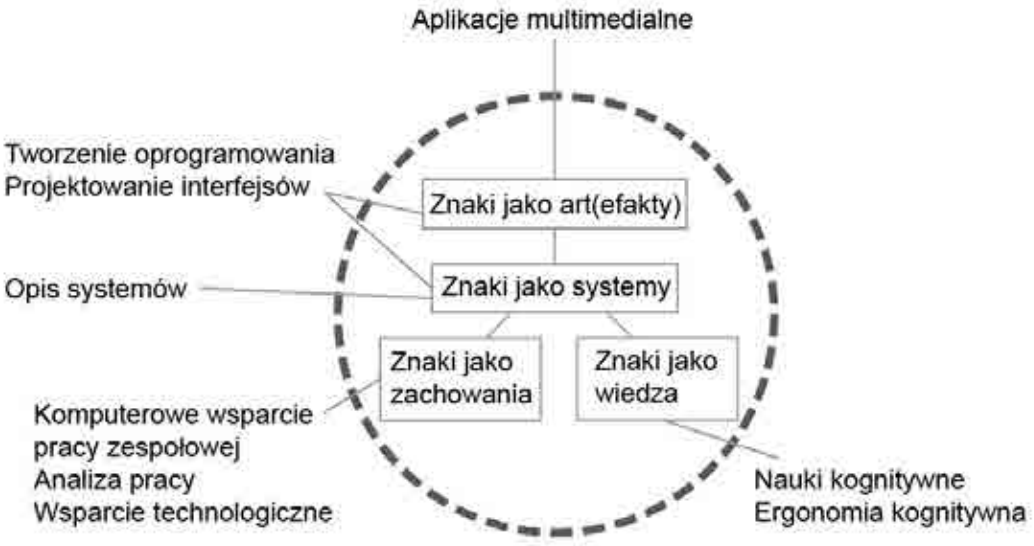
(Goffey, 2008, p. 18)

Autor odwołuje się do bardzo trafnego spostrzeżenia Gillesa Deleuze'a i Felixa Guattari'ego (których analizy – jak sam zauważa – mają miejsce przecież w obszarze pragmatyki języka), że problem z formalną koncepcją algorytmu w maszynie nie polega na tym, że jest on zbyt abstrakcyjny lecz na tym, że jest on nie dość abstrakcyjny – kłopot polega bowiem na ujęciu miejsca algorytmu w procesach, które przekraczają zarówno maszynę jak i człowieka. Otwiera to nasze rozważania na zupełnie nowy obszar, w którym bardzo szeroko rozumiana informatyka stanie się bezpośrednim i współczesnym obszarem zainteresowania humanistyki.

Interfejs między częściami 1:
Od Σ do A: Maszyny semiotyczne.

Od Σ do A 1: Pytanie o „semiotykę komputerową”.

W drugiej części pracy zajmiemy się kwestiami związanymi z budową i sposobami działania oprogramowania komputerowego. Programy – jak się przekonujemy – są bardzo specjalnym sposobem komunikacji. Z jednej strony są one konstruowane stricte językowo – programista pisze kod, z drugiej zaś są całkowicie ukierunkowane na pragmatykę działania, ich skuteczność może i powinna być mierzona w sposób daleko bardziej precyzyjny i wymierny ilościowo niż to ma miejsce w przypadku wypowiedzi języka naturalnego. Nie ulega w każdym razie wątpliwości, że oprogramowanie możemy lokować w uniwersum semiotycznym, do jego fundamentalnych właściwości należy bowiem operowanie na znakach. Kiedy wygłaszamy tego rodzaju stwierdzenie, przedmiot naszego zainteresowania zostaje natychmiast uwikłany w gęstą sieć tradycji związanych z tą dziedziną nauki. Nie chcąc zagłębiać się w konkretne, historyczne teorie znaku i znaczenia, przyjmiemy tu perspektywę globalną, traktując różne podejścia w możliwie szerokim ujęciu:



Ilustracja 7: Mapa semiotyki komputerowej (za: Andersen 1990)

Podejmując taką decyzję, stajemy oczywiście w obliczu ryzyka potraktowania systematycznej semiotyki jako „ogólnej teorii wszystkiego” i dobierania obserwowanych

faktów, tak aby pasowały do systemowej analizy. Jest to problem wszelkich tego rodzaju przedsięwzięć. Jeśli jednak pragniemy w kolejnych częściach pracy przyjrzeć się sposobom działania oprogramowania w ogóle – pojawi się konieczność dokonania wyboru narzędzi badawczych. Logicznie uwarunkowana semiotyka i teorie pokrewne wydają się tu o tyle dobrym wyborem, że przyglądając się kolejnym poziomom funkcjonowania komputerowego uniwersum – od kodu maszynowego, realizowanego na bardzo niskim, technicznym, materialnym poziomie po duże abstrakcje informacyjne i systemowe o charakterze funkcjonalnych ontologii – wszędzie napotkamy znaki i różne ich operacjonalizacje.

Mihai Nadin w swoim wprowadzającym do semiotyki komputerowej artykule pisze:

„Istnieją maszyny zasilane manualnie; inne aktywuje przepływ wody, pary czy grawitacja; jeszcze inne są uruchamiane elektrycznie. Możemy wskazać również biologiczne maszyny, których działanie jest wynikiem procesów biologicznych. (...) Automat jest matematyczną maszyną, która posiada „wejście" (input), zestaw stanów wewnętrznych i wytwarza „wyjście" (output). (...) W dowodzeniu równości między automatami a procesami znakowymi (w ujęciu Pierce'a, ponieważ inne definicje odnoszą się do konkretnych przypadków) istnieje stabilna metodologiczna dyskusja (...). Podsumowując same dowody ustalmy, że Pierce'a definicja może być wyrażona formalnie w następującej postaci:

$S = S(R,O,I,o,i)$ co jest równoważne z $A = A(X,Y,Q,\alpha,\beta)$

gdzie S to proces znakowy będący wynikiem relacji między obiektami, reprezentamenami i interpretantem; A oznacza procesy automatyczne; X i Y kolejno – znaki wejścia i wyjścia; Q – zbiór stanów. Funkcja przejścia i funkcja wyjścia opisują jak wynik jest generowany z pewnego wejścia. Każdy automat jest generatywną semiotyką. Kiedy już równoważność została udowodniona, uzasadnia ona spostrzeżenie wielokrotnie czynione ale nigdy naprawdę nie rozumiane: Komputer jest silnikiem (*engine*) semiotycznym."

(Nadin, 2007, pp. 64-65)

Należy oczywiście zdać sobie sprawę z faktu, że komputer jako maszyna semiotyczna interpretuje znaki jedynie w sensie logicznym. Wielu badaczy, takich jak Peter Bøgh Andersen czy Frieder Nanke zwraca uwagę na konieczność opracowania

koncepcji „znaku algorytmicznego” jako specyficznej, funkcjonalnej podstawy oprogramowania. Chodzi o zwrócenie uwagi na dwoisty sposób funkcjonowania obiektów softwarowych. Po pierwsze istnieją one jako pewne aktywności, zjawiska, wyglądy dla odbiorcy ludzkiego (medialny wymiar oprogramowania), po drugie – zbudowane są ze struktur danych, o których kodowaniu i sposobie funkcjonowania już nieco powiedzieliśmy. Ten drugi, podpowierzchniowy sposób działania komputerowego softwaru jest przeznaczony dla urządzenia do przetwarzania danych, procesora. Jednak skutki działania tegoż, są dostępne dla użytkownika urządzenia za pośrednictwem interfejsu (który znów - jest oprogramowaniem). Ten rodzaj podwójnego kodowania nie jest oczywiście charakterystyczny wyłącznie dla medium komputerowego ale sposób działania „dolnej warstwy” medium, a w szczególności skala i dynamika z jaką odbywają się tam procesy jest bezprecedensowa. Wydaje się to jednym z najdonioślejszych skutków rewolucji cyfrowej w mediach – podstawa bytowa (dane czyli różnice w stanach magnetycznych dysku etc.) i logika działania (algorytm) są całkowicie oderwane od warstwy reprezentacji. Dokładniej rzecz ujmując – istnieją między nimi nadal relacje przyczynowo skutkowe (to jak działa oprogramowanie może być widziane na ekranie) ale mają one charakter całkowicie arbitralny. Praca softwaru może być uwidoczniona (częściowo lub w wybranym zakresie) ale nie musi. Zanim jednak dotrzemy do momentu, kiedy będziemy mogli przyjrzeć się „znakom komputerowym” w postaci fragmentów kodu i ich reprezentacjom na ekranie komputera zapośredniczonym interfejsami (okaże się, że przełożenie jednych na drugie jest zupełnie nietrywialne), poświęćmy jeszcze nieco uwagi działaniu komputera.

Od Σ do A 2: Maszyna syntaktyczna. Język jako model obliczeniowy.

W drugiej połowie lat pięćdziesiątych XX wieku, przeszło trzy dekady po rewolucyjnej pracy Turinga *On computable numbers* Avram Noam Chomski, pracownik naukowy Massachusetts Institute of Technology sformułował wyniki swoich badań nad językami formalnymi. Punktem wyjścia rozważań lingwisty były kwestie związane z akwizycją języka przez dzieci (czyli użytkowników wcześniej niedoświadczonych w użyciu jakiegokolwiek systemu wypowiedzi). Doszedł on do wniosku, że uczenie się języka jest pewną cechą, którą ludzie (i być może nie tylko oni) posiadają gatunkowo (warto podkreślić – znów idea mająca swoje korzenie w filozofii Leibniza). Uczenie się języka przebiega na podstawie posiadanej a priori gramatyki uniwersalnej przez przyswojenie cząstek leksykalnych oraz pewnych parametrów i przełączników na podstawie przykładów.

W ten sposób dochodzi w koncepcji językoznawcy do ścisłego rozróżnienia reguł syntaktycznych i semantycznych rządzących językiem (fonetykę zostawiamy w tym miejscu na boku, ze względu na fakt, że nie leży ona w bezpośrednim obszarze naszych zainteresowań). Wyróżnienie przez Chomsky'ego aparatu służącego nabywaniu zdolności językowej (zwanego LAD: *Language Acquisition Device*) spotkało się z szeroką krytyką ze strony psychologii i psycholingwistyki (znów, nie będziemy tu wnikać w naturę kontrowersji). Z perspektywy studiów nad oprogramowaniem niezwykle istotne pozostaje jednak uczynione w ramach wspomnianego projektu spostrzeżenie, że wszystkie języki naturalne są do siebie podobne strukturalnie ze względu na to, iż w gruncie rzeczy są sparametryzowanymi (przez obcowanie z przypadkami konkretnego języka) gramatykami uniwersalnymi. Taki skonkretyzowany mechanizm (sic!) pozwala na generowanie nieskończonej ilości wypowiedzi dla danego języka (gramatyka generatywna):

„Znajomość języka zakłada niejawną zdolność do rozumienia nieograniczenie wielu zdań. Stąd, aby wygenerować nieograniczenie wielką ilość struktur gramatyka generatywna musi być systemem reguł, które mogłyby się powtarzać. Ów system reguł da się rozłożyć na trzy główne działy: syntaktyczny, fonologiczny i semantyczny"

(Chomsky, 1992, p. 45)

Konsekwencje powyższego stwierdzenia i fakt możliwości formalnego opisu działania GG mają ogromne znaczenie przede wszystkim na gruncie informatyki, językoznawstwa komputerowego oraz badań nad sztuczną inteligencją. Aby przyjrzeć się temu zjawisku zacznijmy od przytoczenia niektórych definicji podstawowych pojęć (za: Gierasimczuk, 2010):

- *Alfabetem* będziemy nazywali skończony i niepusty zbiór symboli. (np. $A = \{a,b\}$);
- *Słowem* będzie skończony ciąg symboli wybranych z pewnego alfabetu (np. „aabbaabbaba");
- Jeśli A jest alfabetem to A^k oznaczać będzie zbiór słów długości k z alfabetu A (np. $\{a,b\} = \{aa,ab,ba,bb\}$). Dla każdego $A^0 = \{\lambda\}$;
- A^* to zbiór wszystkich słów alfabetu A (np. $\{a,b\}^* = \{\lambda,a,b,ab,ba,bb,aaa,...\}$ inaczej $A^* = \bigcup_{n} \in \omega A^n$;
- Zbiór słów z A^* alfabetu A to język; Jeśli A jest naszym alfabetem i jest prawdą, że $L \subseteq A^*$ to L jest językiem nad A^*;

Gramatyka G jest to struktura postaci (A,Σ,S,P) gdzie:

- A to alfabet;
- Σ to zbiór zmiennych;
- $S \in \Sigma$ to symbol początkowy;
- P to skończony zbiór par takich, że $\alpha_i \to \beta_i \ \ (i = 1,...,n)$ oraz $\alpha_i, \beta_i \in (A \cup \Sigma)^*$;
- Dla każdego $\gamma, \gamma' \in (A \cup \Sigma)^*$ jest tak, że γ' jest (w ramach G) **wyprowadzalna bezpośrednio** z γ (inaczej: $\gamma \to \gamma'$) wtw gdy istnieją η_1, η_2 oraz $i = 1,...,n$ takie że $\gamma = \eta_1 \alpha_i \eta_2$ oraz $\gamma' = \eta_1 \beta_i \eta_2$;
- γ' jest w G **wyprowadzalna** z γ (inaczej: $\gamma \to G^* \gamma'$) wtw gdy istnieje ciąg $\gamma_1,...,\gamma_n \in (A \cup \Sigma)^*$ taki, że $\gamma = \gamma_1$, $\gamma' = \gamma_m$ oraz $\gamma_i \to G^* \gamma_{i+1}$ dla $i = 1,...,m-1$;

W końcu:

• Język generowany przez gramatykę G to zbiór $L(G) = \{\gamma \in A^* : S \to G^* \gamma\}$;

Dlaczego rzecz jest dla nas taka ważna? Przyjrzyjmy się działaniu gramatyk generatywnych w praktyce. Jeśli weźmiemy język L zawierający takie wypowiedzi jak „Hiro ma komputer", „Lawrence obsługuje abakus" itd. to jeżeli znamy jego strukturę możemy opisać wywód każdego zdania (generację) za pomocą szeregu produkcji (reguł przepisujących) oraz alfabetu i zbioru symboli nieterminalnych (zmiennych, Σ) czyli mamy:

Język L = {Hiro ma komputer, Lawrence łamie kody, …}
$G = (A,\Sigma,S,P)$ postaci:

A = {Hiro, Lawrence, ma, obsługuje, komputer, abakus}
$\Sigma = \{S, A, B\}$
P = { 1. S → Hiro A
2. S → Lawrence A
3. A → ma B
4. A → obsługuje B
5. B → komputer
6. B → abakus }
$S \in N$

Generacja zdania „Hiro ma komputer” będzie wyglądała następująco:

$S^1 \rightarrow$ Hiro $A^3 \rightarrow$ Hiro $ma^5 \rightarrow$ Hiro ma komputer

Gramatyki mogą opisywać też języki nieskończone. Weźmy przykład:

$$L_a = \{x^n, y^m, n \geq 1, m \geq 1\}; G_a = (A, \Sigma, S, P_a) : L_a = L(G_a)$$
$$A = \{x, y\}; \Sigma = \{S, A, B\}, P_a = \{1.S \rightarrow xA; 2.A \rightarrow xA; 3.A \rightarrow yB; 4.B \rightarrow yB; 5.B \rightarrow y\}$$

W tak zdefiniowanej sytuacji możemy wygenerować zdanie dowolnej długości ponieważ produkcje dają się stosować rekurencyjnie zaś jedynym warunkiem powstania poprawnego zdania należącego do L_a jest zawartość co najmniej jednego x i jednego y (czyli najkrótszym zdaniem generowanym przez tą gramatykę będzie „xy”).

W tym momencie możemy wprowadzić zaproponowany przez Chomsky’ego podział na cztery klasy języków formalnych. Dają się one rozróżniać ze względu na „kształt” generowanych przez nie produkcji na:

1. Gramatyki rekurencyjnie przeliczalne.
   Ich produkcje są postaci ά→β (ά, β to dowolne słowa).
2. Gramatyki kontekstowe. Ich produkcje są postaci α A β → α γ β (ά, β to dowolne słowa; A to symbol nieterminalny, γ to niepuste słowo)
3. Gramatyki bezkontekstowe. Ich produkcje mają postać A → γ (po lewej stronie reguł produkcji mamy pojedynczy symbol nieterminalny, po prawej dowolne słowa)
4. Gramatyki regularne. Ich produkcje maja postać A → ά lub A → aB (po jednej stronie mają pojedynczy symbol nieterminalny, po drugiej słowa zawierające co najwyżej jeden symbol nieterminalny)

Jeśli przez L_i oznaczymy rodzinę wszystkich języków danego typu (i = 0, 1, 2, 3) to wybrany z nich jest typu i wtedy i tylko wtedy, gdy jest rozpoznawany przez automat z odpowiedniej rodziny. W ten sposób języki typu 0 (rekurencyjnie przeliczalne) są rozpoznawane przez maszynę Turinga, 1 (kontekstowe) przez automat liniowo ograniczony, 2 (bezkontekstowe) automat ze stosem, zaś 3 (regularne) – deterministyczny

automat skończony. Wynika stąd, że – zgodnie z tym co na temat koncepcji Alana Turinga zostało już stwierdzone – granica rozstrzygalności języków przebiega między typem 0 i 1.

W ramach rozważanej perspektywy maszyna Turinga służy nam do „akceptowania" pewnych rodzajów języków, tymczasem zagadnieniem naprawdę nas interesującym jest algorytmiczne (na wyższym poziomie – softwarowe) rozwiązywanie pewnych klas problemów. W związku z tym należy przyjąć język jako jeden z rodzajów kodowania wprowadzanego dla skutecznego rozwiązania postawionego problemu. Trzeba przy tym pamiętać, że nierozstrzygalność algorytmiczna w tym przypadku nie oznacza, że żadnego problemu z danej klasy nie da się rozwiązać lecz raczej fakt, że nie mamy uniwersalnej metody rozwiązywania wszystkich przypadków ze wskazanego obszaru.

Od Σ do A 3: Dwa wektory.

W tym punkcie naszych rozważań napotykamy na rozstajne drogi. Jedna z nich prowadzi w głąb szczegółowych rozważań z dziedziny matematyki, teorii obliczeń, wiąże się też silnie ze szczegółowymi kwestiami implementacji naszych rozważań w niesłychanie wyrafinowane układy elektroniczne. Druga ze ścieżek wiedzie powoli wzwyż – ku interfejsom i „powierzchni ekranu" – powoli, po kolejnych stopniach abstrakcji wznosząc się od fundamentalnych rozważań na temat właściwości najmniejszych cząstek informacji ku coraz bardziej złożonym strukturom z postępującą wiernością oddającym konstrukcje języka naturalnego (języki programowania wysokiego poziomu) oraz nasz, ludzki sposób postrzegania świata (ontologie semantyczne i obiektowe). Z punktu widzenia prowadzonych na stronach niniejszej dysertacji rozważań znacznie bardziej interesująca wydaje się druga z opcji. Podążymy więc tą drogą – aby wychodząc od najprostszych lecz najbliższych charakterystyce automatu i maszyny Turinga konstruktów (język maszynowy) zmierzać w stronę horyzontu wypełnionego bliższymi intuicyjnemu przedmiotowi zainteresowania medioznawcy obiektami. Nie wolno nam jednak zapominać o podłożu, matematyczno-elektronicznej podstawie, która stanowi nieusuwalny grunt wszelkiego rodzaju omawianych w dalszej części pracy zjawisk i procesów.

Αναλυτικά. Analityka. „Terminem tym Arystoteles oznacza to, co później zostanie nazwane logiką, albo przynajmniej główną jej część. Analityka oznacza metodę, za pomocą której, *wychodząc od danego wniosku, rozkłada się go – właśnie przez analizę – na elementy, z których on wypływa, to znaczy na przesłanki, z których wynika, uwidaczniając w ten sposób związki łączące przesłanki z wnioskiem.*" (Reale 2005, p. 23)

Część 2: Analityka (Αναλυτικά).

Αναλυτικά 1: Język maszynowy.

„Może nie wiecie, w tych dekadenckich czasach lekkiego piwa, ręcznych kalkulatorów i »przyjaznego« softwaru ale kiedyś, w Starych Dobrych Czasach, kiedy pojęcie »oprogramowania« brzmiało zabawnie, a Prawdziwe Komputery były zrobione z bębnów i próżniowych rur, Prawdziwi Programiści pisali w kodzie maszynowym. Nie Fortranie, Nie RATFOR'rze. Nawet nie w języku assemblera. Kodzie Maszynowym. Surowe, pozbawione ozdobników, instrukcje w liczbach heksadecymalnych. Bezpośrednio."

The Story of Mel, 21 maja 1983[6]

„Każda maszyna posiada listę czynności, których wykonanie możesz jej nakazać. Czasami ta lista jest krótka. Z moim czajnikiem elektrycznym mogę zrobić tylko dwie rzeczy: włączyć go i wyłączyć. (...) Jak każda inna maszyna, komputer ma listę czynności, które może wykonać. Na przykład każdemu komputerowi można nakazać dodanie do siebie dwóch liczb. Kompletna lista zadań, które komputer może wykonać, nazywa się językiem maszynowym."

(Graham, 2004, p. 179)

Naszą podróż „drugą ścieżką" – w stronę świata wysokopoziomowych abstrakcji musimy zacząć (znów) od rudymentów. Nie mniej jednak będą to już podstawy innego rodzaju, wywodzące się co prawda z opisanych na poprzednich stronach źródeł ale tym razem sprowadzone już do poziomu konkretnej implementacji. Z paradygmatu teoretyczno-historycznego przechodzimy do rzeczywistości faktycznego oprogramowania, działającego i aktywnego w świecie.

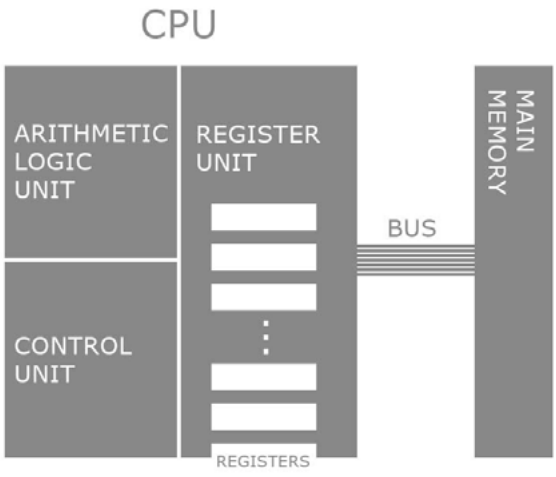
[6] „The Story of Mel" to historia pewnego archetypowego programisty, element sieciowego i hakerskiego folkloru, funkcjonująca w Sieci w różnych formach i odmianach. Podobno pierwsza wersja została opublikowana przez użytkownika *utastro!nather* na Usenecie w maju 1983 roku. Można ją przeczytać np. tu: http://pbm.com/~lindahl/mel.html (dostęp 1.01.2014)

A 1.1 Procesor.

W poprzedniej części pracy rozważaliśmy między innymi kwestie związane z definicją, charakterystyką i konceptualnymi źródłami pojęcia informacji, danych a także teoretycznymi podstawami tego jak mogą być one przetwarzane. Żeby obraz ten uczynić pełnym musimy pochylić się jeszcze nad tym jak przytaczane teoretyczne koncepcje są realizowane w faktycznych urządzeniach – hardwarze.

Podstawową jednostką komputera, służącą do przetwarzania danych jest procesor, zwany też CPU (*Central Processing Unit*). Historia tego urządzenia sięga czasów Turinga i jego „bomb" – wielkich maszyn służących do realizowania bardzo konkretnych algorytmów związanych z proceduralnym łamaniem niemieckich szyfrów w czasie II Wojny Światowej. Ikoniczne, nieco późniejsze protokomputery w rodzaju Collosusów lub ENIAC'a – potężne szafy wypełnione lampami elektronowymi bardzo szybko zastąpione zostały urządzeniami na bazie krzemowych tranzystorów, co doprowadziło do radykalnego zagęszczenia elementów przy równoczesnej miniaturyzacji. Dzisiejsze mikroprocesory zawierają setki milionów tranzystorów „wygrawerowanych" metodą fotolitografii na warstwach półprzewodnikowych w monokryształe krzemu.

CPU zasadniczo składa się z trzech części - jednostki arytmetycznej/logicznej (temat działania bramek logicznych był już poruszany we fragmencie „Σ 2.Apendyks: Jakiego kodu użyć?"), jednostki sterowania oraz jednostki zawierającej rejestry służące do przechowywania danych będących przedmiotem manipulacji procesora.



Ilustracja: Budowa CPU (za: Brookshear, 2012, p. 74)

Najogólniej ujęta zasada działania procesora polega więc na tym, że jednostka sterująca podaje dane z pamięci do rejestrów ogólnego przeznaczenia, informuje jednostkę arytmetyczno-logiczną, które rejestry przechowują istotne dane, aktywuje właściwe obwody w jednostce arytmetyczno-logicznej i wskazuje jej do których rejestrów powinny trafić rezultaty wykonanych operacji. Na potrzeby transmitowania wzorców bitowych CPU i pamięć komputera są połączone „szyną" (magistralą/bus) – za jej pomocą jednostka centralna wydobywa (odczytuje) dane z wybranej komórki pamięci. W taki sam sposób CPU umieszcza (zapisuje) dane w pamięci – podając adres docelowej komórki i dane, które mają być zachowane wraz z właściwym sygnałem elektronicznym informującym pamięć, że ma przechować dane do niej przesyłane.

Przykładowa operacja sumowania wartości przechowywanych w pamięci może więc wyglądać następująco:

1. Pobierz pierwszą z wartości, które mają być dodane z pamięci i umieść ją w rejestrze;
2. Pobierz drugą wartość, która ma być sumowana z pamięci i umieść w innym rejestrze;
3. Aktywuj obwód dodawania z rejestrami z kroku 1 i 2 jako wejściem oraz jeszcze inny rejestr przeznaczony do przechowania wyniku;
4. Zapisz wynik w pamięci;
5. Stop;

(za: Brookshear, 2012, pp. 74-75)

Historycznie wykonywanie tych operacji odbywało się przez przełączanie odpowiednich kabli (łącznie „jumperami" par otworów) na tablicy rozdzielczej. Dziś odpowiednio zaprogramowany procesor wykonuje takie operacje z zawrotną częstotliwością (wielokrotnością gigaherców czyli $x*10^9$ razy na sekundę). Oczywiście należy pamiętać, że poszczególne architektury komputerowe różnią się niuansami działania całego systemu oraz, że dzisiejsze komputery operują na bardziej złożonych, wspomaganych układach – zawierających np. pamięć podręczną (cache) czy wykonując operacje równolegle w kilku rdzeniach lub rozkładając je na wiele procesorów jednocześnie. Nie mniej zasada działania pozostaje ogólnie taka sama.

Niesłychanie istotnym momentem w rozwoju technik komputerowych było uświadomienie sobie przez twórców, że oprogramowanie – podobnie jak dane – może być przechowywane w pamięci. Możemy tak zaprojektować jednostkę sterującą, że będzie potrafiła wydobyć z pamięci nie tylko dane ale również zinterpretować je jako instrukcje i wykonać je. Oprogramowanie może więc być zmieniane przez znacznie łatwiejsze od modyfikowania CPU zmiany w komputerowej pamięci. To przesunięcie i zlikwidowanie rozróżnienia przydziału: dane w pamięci/programy w CPU było początkowo trudne koncepcyjnie. Kiedy jednak udało się go dokonać, otworzyło to zupełnie nowy wymiar dla oprogramowania.

A 1.2 Instrukcja po instrukcji. Bit po bicie.

W związku z opracowaniem koncepcji programu w pamięci – wiązanej często, acz nie do końca słusznie, z pojawieniem się tzw. architektury von Neumanna (lub architektury Princeton) pojawiła się konieczność przygotowania wzorca kodowania instrukcji, które mogłyby być przechowywane w pamięci. Taki zbiór (instrukcji i systemu kodowania) nazywamy językiem maszynowym zaś o poszczególnych jego poleceniach mówimy, że mają one miejsce na poziomie maszynowym.

Lista poleceń kodu maszynowego musi być bardzo krótka. Wszelkiego rodzaju jej rozwinięcia są oczywiście możliwe ale służą one raczej ułatwieniu konceptualizacji złożonych problemów i dają się zawsze redukować do komend najprostszych – tych zaś są trzy grupy:

1. Transferu danych;
2. Logiki/arytmetyki;
3. Sterowania;

Ad. 1
Polecenia należące do tej grupy mają za zadanie kopiowanie (klonowanie) danych z jednej lokacji w inną. Szczególnym przypadkiem jest transfer odbywający się między CPU a pamięcią. Mamy tu do czynienia z instrukcjami LOAD (komórka pamięci – rejestr) i STORE (rejestr – komórka pamięci). Dodatkowo możemy wyróżnić jeszcze instrukcje I/O służące do obsługi urządzeń wejścia/wyjścia.

Ad. 2

Jednostka arytmetyczno/logiczna służy do wykonywania znanych nam już z poprzednich części pracy operacji boolowskich (AND, OR, XOR). Może ona też wykonywać proste operacje przenoszenia zawartości rejestru (SHIFT, ROTATE).

Ad. 3

Grupa instrukcji sterowania stanowi raczej środek wykonywania poleceń niż manipulowania danymi. Znajdziemy tu takie instrukcje jak JUMP (BRANCH), które pozwalają na wykonywanie pewnych operacji „poza kolejką". Tego typu instrukcje mogą być wykonywane warunkowo np. w celu uniknięcia sytuacji niedopuszczalnych (jak dzielenie przez zero).

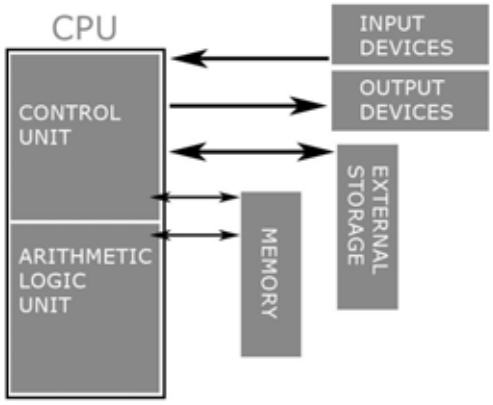
Przykładowa operacja dzielenia wartości przechowywanych w pamięci:

- LOAD do rejestru wartość z pamięci;
- LOAD drugiej wartości z pamięci do rejestru;
- Jeśli wartość w 1 wynosi zero JUMP do 6;
- Podziel wartość pierwszego rejestru przez wartość drugiego rejestru i zapisz wynik do trzeciego rejestru;
- STORE zawartości trzeciego rejestru do pamięci;
- STOP;

(za: Brookshear, 2012, pp. 78-79)

Na marginesie : architektura von Neumanna / Princeton

Jeden z pierwszych rodzajów architektury komputera, dzieło Johna von Neumanna, Johna Mauchly'ego oraz Johna P. Eckerta. Opracowany w 1945 roku model charakteryzuje się tym, że dane przechowywane są razem z instrukcjami zaś komputer zbudowany jest z **pamięci** (dane i instrukcje), **jednostki sterującej** (pobieranie z pamięci, przetwarzanie), **jednostki arytmetyczno-logicznej** (operacje), **urządzeń wejścia/ wyjścia** (interfejsy).



Uproszczony schemat modelu Princeton

Ostatnią kwestią wymagającą wyjaśnienia w tym rozdziale jest sposób w jaki kodowane są instrukcje w pamięci. Zakodowana wersja instrukcji maszynowej składa się z dwóch części: kodu operacji (*opcode*) i pola operanda. Wzorzec bitowy znajdujący się w polu op-code wskazuje, którą z elementarnych operacji (np. STORE, XOR, JUMP) wywołuje instrukcja. Wzorzec znajdujący się w polu operanda dostarcza bardziej szczegółowych informacji na temat operacji oznaczonej w op-code. Na przykład w operacji STORE wskazuje, który z rejestrów zawiera dane przeznaczone do przechowania i która komórka pamięci ma otrzymać dane. Cały język maszynowy składa się ze zbioru kilkunastu podstawowych instrukcji, z których każda jest zakodowana przy użyciu pewnej ilości bitów (8, 16, 32, 64) reprezentowanych przez liczby heksadecymalne (układu szesnastkowego):

Op code	Operand			
0 0 1 1	0 1 0 1	1 0 1 0	0 1 1 1	Wzorzec bitowy (16 bitów)
3	5	A	7	Forma heksadecymalna (4 znaki)

Przykład: 16 bitowa instrukcja z przepisaniem na formę heksadecymalną
(za: Brookshear, 2012, pp. 90-81)

Instrukcja 35A7 wyrażona w heksach tłumaczy się na wyrażenie „STORE wzorzec bitowy w registrze 5 do komórki pamięci, której adres to A7”. Kod heksadecymalny ułatwia odczytanie polecenia, które w systemie binarnym brzmi: 0011010110100111.

Komputer wykonuje program zapisany w pamięci przez kopiowanie z niej instrukcji do CPU w ramach której mamy do czynienia z rejestrem instrukcji i licznikiem programu. Pierwszy z nich jest używany do przechowywania instrukcji, które są wykonywane – drugi zaś zawiera adres kolejnej instrukcji do wykonania. CPU pracuje w składającym się z trzech kroków cyklu maszynowym polegającym na wykonywaniu operacji:

1. FETCH. Kolejna instrukcja pobierana jest z pamięci (zgodnie ze wskazaniem licznika) i inkrementowana do niego;
2. DECODE. Wzorzec bitowy jest dekodowany w rejestrze instrukcji;
3. EXECUTE. Akcja wymagana przez instrukcję jest wykonywana;

Ilość wykonywanych przez urządzenie cykli na sekundę (częstotliwość cyklu zegarowego) to jeden z najważniejszych parametrów określających szybkość działania procesora. Określa się współcześnie z reguły w gigahercach czyli miliardach operacji wykonywanych na sekundę.

Jak łatwo zauważyć – przejście od języka maszynowego do asemblera odbywa się bardzo subtelnie. Jak pisze Paul Graham:

„Po wynalezieniu komputerów wszystkie programy musiały być pisane jako ciągi instrukcji języka maszynowego. Wkrótce potem zaczęto je pisać w odrobinę wygodniejszej formie, nazwanej asemblerem. W asemblerze do dyspozycji pozostają te same polecenia, ale możesz dla nich używać nazw przyjaźniejszych dla programisty. Zamiast odwoływać się

do instrukcji *add* jako do 11001101, co jest jej nazwą w języku maszynowym, możesz po prostu napisać *add*."

(Graham, 2004, p. 179)

Dla nas tematyka programowania i oprogramowania staje się tak naprawdę interesująca kiedy dla wspomnianej przez autora wygody polecenia sterujące pracą procesora i przetwarzaniem danych zaczynamy ubierać w coraz bardziej rozbudowane stopnie abstrakcji. Zaczynamy układać kod na podobieństwo świata. Dokładniej zaś rzecz ujmując – na nasze sposoby jego rozumienia i konceptualizowania.

Αναλυτικά 2:
Struktury danych + algorytmy = programy[7] (*software*).

„Trzy są czynności, w których umysł wykonuje swą władzę nad ideami prostymi: 1. łączenie pewnej liczby idei prostych w jedną złożoną; tak powstają wszystkie idee złożone. 2. zestawienie łączne dwu idei, prostych lub złożonych, i to takie zestawienie, aby można je było na raz ogarnąć wzrokiem, nie stapiając ich w jedno; na tej drodze dochodzi umysł do wszelkich swych idei stosunków. 3. oddzielanie lub odrywanie idei od wszelkich innych, jakie im towarzyszą faktycznie w świadomości; nazywa się to abstrahowaniem i tak powstają wszystkie idee ogólne”

(Locke, 1955, p. XII:1)

„Algorytm = Logic + Control”

(Kowalski, 1979)

W poprzednich rozdziałach nasze rozważania dotyczyły głównie informacyjno-obliczeniowej natury urządzeń komputerowych. Zastanawialiśmy się przede wszystkim nad najbardziej podstawowymi operacjami jakie zachodzą wewnątrz maszyn cyfrowych. Interesowała nas ich logika – źródło wszelkich rozważań na temat nowego, elektronicznego świata. Doszliśmy do podstawowego pojęcia bitu i konsekwencji związanych z posługiwaniem się tą najmniejszą jednostką informacji. Próbując jednak spojrzeć na świat komputerowego softwaru z pełnej perspektywy widzimy przede wszystkim niesłychanie złożone oprogramowanie, modelujące i naśladujące procesy, o których mówimy, że są „wysokiego rzędu” – interpretację zachowań, podejmowanie decyzji, myślenie. Doszukanie się w prostej pracy procesora, bitowych przełącznikach czy elementarnej logice gramatyk generatywnych śladów operacji, o których myślimy jako o niesłychanie złożonych i wyrafinowanych nie jest rzeczą łatwą... lecz wykonalną. Oto bowiem przed nami fakt – wszystkie komputery, to znaczy również te, które sterują ruchem pojazdów kosmicznych, przetwarzają dane miliardów podróżujących turystów, zawiadują systemami telekomunikacyjnymi, wygrywają z ludźmi w szachy - są w ścisłym sensie implementacjami Maszyny Turinga. Ich praca daje się sprowadzić do działania opisanego przez Brytyjczyka modelu. Niklaus Wirth pisze: „W większości jej zastosowań (nowoczesnej maszyny cyfrowej – dop. J.K.A.) najważniejsze znaczenie ma możliwość

[7] Tytuł rozdziału nawiązuje do tytułu podręcznika: (Writh, 2000)

przechowywania i dostępu do dużych partii informacji (...), które mają być przetwarzane, reprezentują (one – dop. J.K.A.) – w pewnym sensie – abstrakcyjny model części świata rzeczywistego." (Writh, 2000, p. 17). Ów zadziwiający fakt jest możliwy dzięki temu, że proste operacje logiczne i bardzo małe porcje informacji nauczyliśmy się układać w bardzo złożone i wyrafinowane struktury oraz przetwarzać je z ogromną wydajnością. Logika pracy komputera jest więc logiką Maszyny Turinga ale ujętą w szersze ramy algorytmów i struktur danych. Przyjrzyjmy się czym są wspomniane zjawiska aby zrozumieć co to takiego *software*.

A 2.1 Struktury danych.

W informatyce strukturami danych nazywamy takie sposoby przechowywania i organizowania porcji informacji, które sprzyjają ich efektywnemu wykorzystaniu w ramach projektowanego rozwiązania. Różne z tych form udostępniają zawartość zróżnicowanym typom aplikacji, stąd wybór odpowiedniej reprezentacji danych nie jest trywialny również z punktu widzenia języków programowania. W tym podrozdziale postaramy się wyliczyć niektóre podstawowe struktury i ich właściwości posługując się do tego celu przykładami z języka Processing jako przeznaczonego i szczególnie nadającego się do kreatywnego stosowania w sztuce nowych mediów i różnego rodzaju projektach spod znaku *digital humanities.* Dodatkowo, w późniejszych częściach pracy będę się od czasu do czasu wspierał przykładami z JavaScriptu oraz C++ i Javy jako jednych z najpopularniejszych aktualnie języków programowania na świecie (samym językom i ich właściwościom poświęcony został rozdział A 5 tej dysertacji).

Język programowania Processing

Processing - jak piszą jego twórcy stronie internetowej projektu (www.processing.org) - to język programowania, darmowe zintegrowane środowisko developerskie (IDE) oraz społeczność on-line stworzone jako narzędzie dla twórców sztuki elektronicznej.

Processing powstał w ramach Aesthetic and Computation Group (ACG) w przy MIT Media Lab. Jego oryginalnymi twórcami są Ben Fry i Casey Reas ale projekt „uwolnił się" spod opieki autorów i jest rozwijany przez społeczność.

Język pozwala na pisanie kodu zarówno w paradygmacie proceduralnym jak i obiektowym oraz jest multiplatformowy.

Cechą charakterystyczną Processingu jest nastawienie na uproszczenie złożonych konstrukcji programistycznych oraz szybkie uzyskiwanie efektów wizualnych lecz bez pozbawiania narzędzia właściwości „porządnego" języka programowania – pomimo bowiem swojego edukacyjnego rodowodu rozwinął się on do postaci pełnowartościowego narzędzia programistycznego.

Tablice (array) i tablice mieszające (hash table)

Pierwszym i zarazem jednym z najpopularniejszych typów danych jest tablica (ang. array). Funkcjonuje ona zarówno jako struktura danych jak i kontener, w którym poszczególne komórki są dostępne za pomocą kluczy (nazw, numerów). Tablice mogą być jednowymiarowe (ciąg), dwuwymiarowe (macierz) lub wielowymiarowe. W języku Processing tablice inicjuje się poleceniem [ ] podając nazwę tablicy oraz jej strukturę i typ przechowywanych informacji np.:

```
int[] liczba = new int[2]
```

tworzy jednowymiarową tablicę o dwóch elementach i nazwie „liczba", zawierającą liczby całkowite *(integer)*

```
liczba [0] = 1; liczba [1] = 2
```

przypisuje elementowi tablicy „liczba" o indeksie 0 wartość 1 i elementowi o indeksie 1 wartość 2.

Oczywiście na tablicach możemy wykonywać bardziej złożone operacje. Realizować działania arytmetyczne na ich zawartości, umieszczać w pętlach etc.

Krotki [tuple, record]

To struktury danych będące realizacjami n-ki matematycznej, czyli pewnego uporządkowanego ciągu wartości. Przykłady krotek to np.

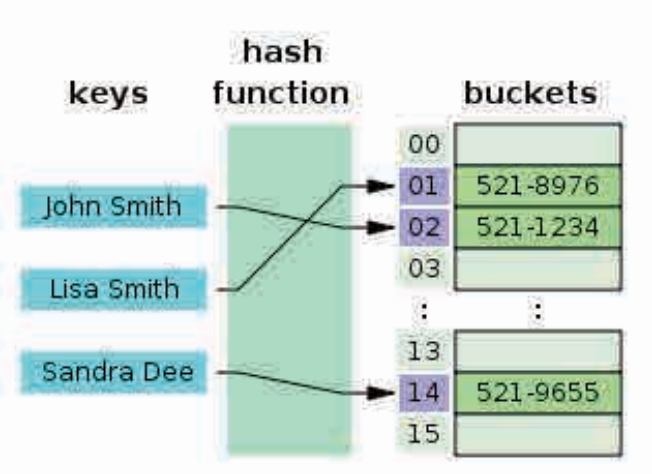
```
Zwierz = („kot”, „mainecoon”, 5)
```

Do takiej konstrukcji możemy odwołać się przez podanie jej indeksu. Na krotkach również możemy realizować operacje logiczne i arytmetyczne. W języku Processing taką strukturę możemy sobie zorganizować poprzez stworzenie klasy z przypisywaną listą parametrów np.:

```
class xy {
          float x,y; xy(float _x, float _y)
          {x=_x; y=_y;}}
```

Tablice mieszające [hash table]

Ta struktura danych daje możliwość realizowania układów w rodzaju tablic asocjacyjnych, które służą do przechowywania informacji szybkiego dostępu. Najprościej rzecz ujmując hash table działa podobnie do zwykłej tablicy z tym, że umożliwia stosowanie funkcji mieszających (czasami mówi się haszujących), które łączą indeks tablicy z zadanym kluczem.



Ilustracja 9: Zastosowanie tablicy haszującej na przykładzie książki telefonicznej.[8]

[8] źródło: http://upload.wikimedia.org/wikipedia/commons/thumb/7/7d/Hash_table_3_1_1_0_1_0_0_SP.svg/315px-Hash_table_3_1_1_0_1_0_0_SP.svg.png (dostęp: 1.03.2014).

Reprezentacja grafu [graph]

Graf stanowi specyficzny sposób uporządkowania i opisu relacji między elementami. Najczęściej pojawia się tu użycie macierzy i listy sąsiedztwa. Formalizacja wygląda następująco: jeśli $G =< V, E >$ to graf, V zbiór wierzchołków, E zbiór krawędzi - to możemy taki układ indeksować przypisując wierzchołkowi $i \in 0...|V|-1$. Zbiór wierzchołków będzie miał postać: $v_0,...,v_{|v|-1}$. Wierzchołki mogą być reprezentowane za pomocą różnorodnych struktur danych. Jest to bardzo wygodna i łatwa do zoperacjonalizowania koncepcja reprezentowanie założonych relacji.

W Processingu możemy zdefiniować graf za pomocą klasy:

```
g = new Graf();
```

Następnie jego wierzchołki i krawędzie za pomocą tablic (array):

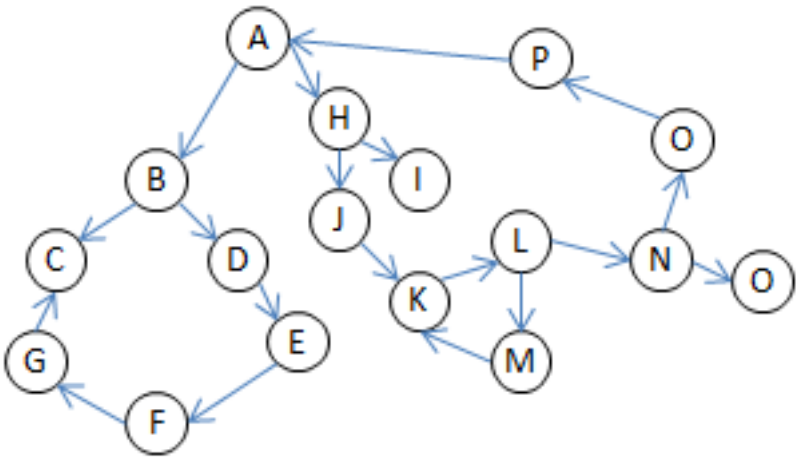
```
Node[] nodes = new Node[numerOfNodes]
```

A dalej wywoływać je do dalszego przetwarzania za pomocą funkcji w rodzaju

```
Node[] nodes = przypisane.wartosci[x][y].getAdjacentNodes(1,0);
```

Oczywiście elementy tablicy trzeba najpierw wygenerować za pomocą np. pętli *for*.

Dzięki podobnym operacjom możemy przedstawiać nasze dane w postaci grafu i generować ich wizualizacje np. takie jak na ilustracji poniżej.

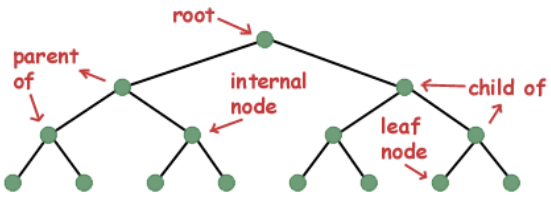


Ilustracja 10: Przykładowa struktura grafowa[9]

[9] źródło: http://stackoverflow.com/questions/10874024/data-structure-and-algorithm-for-circular-graph (dostęp 1.03.2014)

Drzewo [tree]

Drzewo to struktura danych, która mając swoje źródło w matematyce nadaje się bardzo dobrze do reprezentowania hierarchii dowolnych elementów (danych). Podobnie jak graf składa się ona z wierzchołków (węzłów/nodes) oraz krawędzi, które łączą je między sobą. W niepustym drzewie wyróżniany jest jeden element – korzeń. Poziom węzła jest określany przez ilość krawędzi od niego do korzenia. Wysokość jest wyznaczana przez odległość od korzenia do najdalszej krawędzi (gałęzi). Wierzchołki niższe w hierarchii są nazywane dziećmi węzła, jeśli takowy nie posiada elementów niższych – jest liściem.



Ilustracja 11: Przykładowa struktura drzewiasta z opisem[10]

Obiekt [object]

To jedno z najważniejszych pojęć dla naszych dalszych rozważań. Dokładne jego wyjaśnienie znajduje się w rozdziale dotyczącym paradygmatów oprogramowania (dokładnie zaś kiedy mowa jest właśnie o paradygmacie obiektowym [*object oriented paradigm*][11]). Na potrzeby aktualnego wywodu wystarczy opisanie obiektu w sensie programistycznym jako struktury zawierającej dane oraz metody rozumiane jako procedury służące do realizowania konkretnych zadań. Obiekty są swego rodzaju samodzielnymi „bytami” w specyficznej ontologii języków programowania. Waga tego pojęcia dla dalszej części rozważań poświęconych metodom konstruowania i sposobom działania oprogramowania polega na specyficznym zwrocie jaki dokonał się za sprawą OOP. Chodzi mianowicie o fakt, że programy w takim ujęciu nie są już tylko nierozłącznymi zbiorami danych i procedur (do manipulacji danymi) lecz zyskują swoistą autonomię, stają się elementami „świata” konstruowanego za pomocą opisu informatycznego.

[10] źródło: http://img.sparknotes.com/figures/B/becc4efefde067dce51a326cca23c5f0/treedefinition.gif (dostęp 1.03.2014)

[11] vide: *A 4 Paradygmaty programowania*

A 2.1.1 Przykłady prostych programów artystycznych i narzędzi humanistyki cyfrowej wykorzystujących różne struktury danych

Przypadek pierwszy: programowanie *Gry w Życie* przy użyciu TABLIC (Processing)

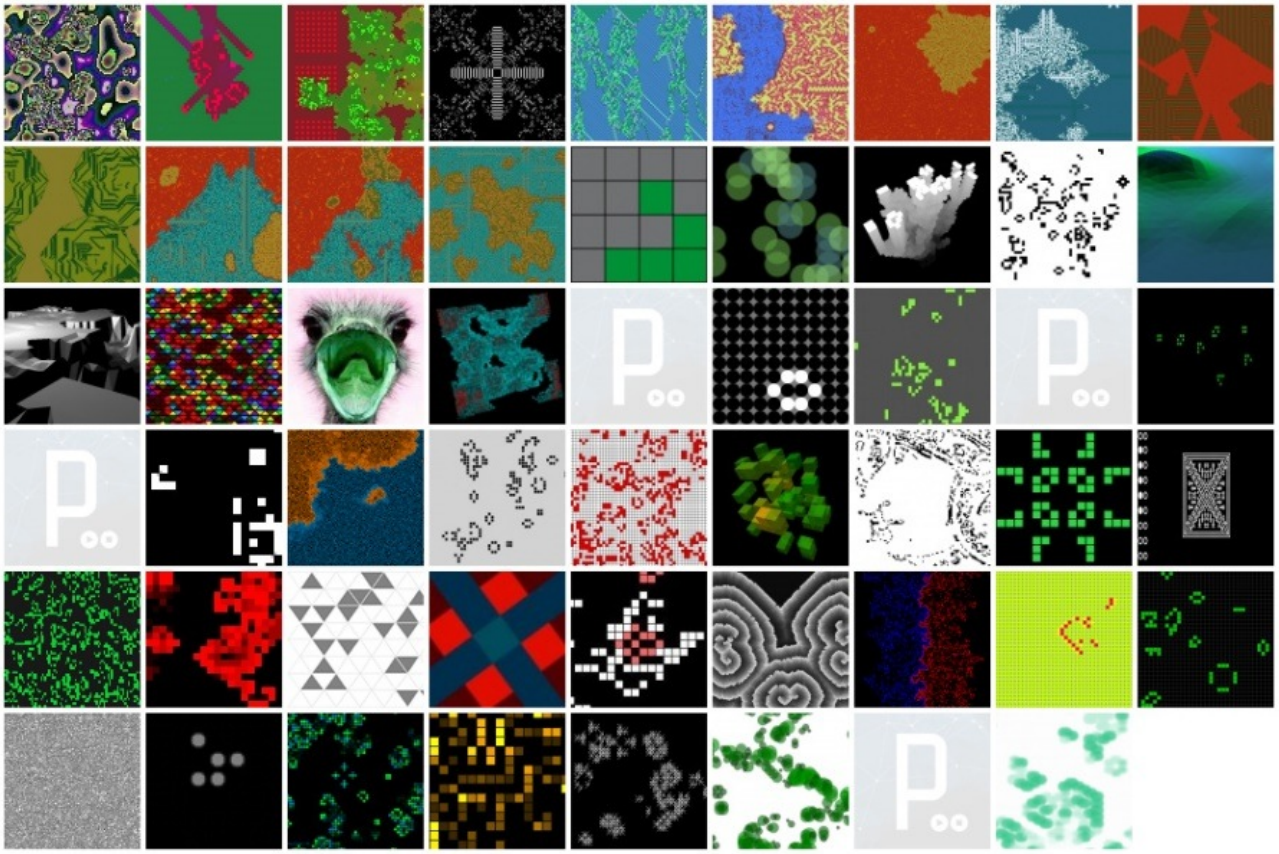
Gra w Życie to jeden z najbardziej znanych przykładów automatu komórkowego opracowany przez brytyjskiego matematyka Johna Conwaya w latach siedemdziesiątych ubiegłego stulecia. Koncepcja ta nazywana jest „grą" z powodu tego, że implementuje się w niej pewien zbiór reguł, który może być interpretowany jako mechanika rozgrywki. Jest to jednak przypadek gry bez gracza (*zero-player game*) ponieważ nie ma w nim miejsca na świadomy podmiot i jego aktywne działanie. W *Grze w Życie* generuje się warunki początkowe układu, a następnie rozgrywka przebiega w sposób automatyczny. Zasady *Gry* mogą być formułowane w kilku stopniach złożoności jednak podstawowy model wygląda następująco:

Dana jest „plansza" podzielona prostokątną siatką. Pole siatki może przyjmować wartość „żywa" (1) lub „martwa" (0). Rozgrywka przebiega zgodnie z regułami dotyczącymi sąsiedztwa:

- każda żywa komórka z mniej niż dwoma żywymi sąsiadami umiera (z samotności);
- każda żywa komórka z dwoma lub więcej sąsiadami żyje do kolejnej generacji (kolejki gry to pokolenia);
- każda żywa komórka z więcej niż trzema żywymi sąsiadami umiera (z przeludnienia);
- każda martwa komórka z dokładnie trzema żywymi sąsiadami staje się żywa (mechanizm reprodukcji);

Niezwykłość *Gry w Życie* polega na tym, że w prostym układzie sterowanym kilkoma regułami dają się zaobserwować niesłychanie nieraz wyrafinowane procesy, uderzające w przypadku wizualizacji swoim „organicznym" charakterem. Pojawiają się różnego rodzaju stabilne lub dynamiczne układy („organizmy") z których wiele ma swoją nazwę. Należą do nich zachowujące swój kształt - „klocek", „łódź", „kryształ" oraz zmieniające się okresowo oscylatory (takie jak „blinker", „żabka") oraz poruszające się lub wygasające po pewnej liczbie pokoleń zjawiska („diehard" umierający po 130 pokoleniach, czy ruchome statki w rodzaju „glidera"). *Gra w Życie* stała się źródłem wielu interesujących

koncepcji symulacyjnych oraz swego rodzaju symbolem emergentnej kreatywności. *Game of Life* znajduje zastosowanie w wielu projektach wizualizacyjnych i artystycznych. Warto wspomnieć, że jeden z układów – „glider" („szybowiec") stał się nieoficjalnym lecz powszechnie rozpoznawalnym symbolem ruchu hackerskiego (zaproponowanym przez Erica Raymonda).



Ilustracja 12: Przykłady projektów artystycznych opartych na koncepcji Conwaya zrealizowanych w środowisku Processing[12]

.

Najłatwiejszym, choć oczywiście nie jedynym sposobem na zaimplementowanie *Gry w Życie* za pomocą języka Processing jest użycie opisywanych w tym rozdziale struktur w postaci tablic. Na potrzeby przykładu możemy posłużyć się tablicą dwuwymiarową, której zawartość będzie wyrażana wartościami boolowskimi („żywe", „martwe").

```
int wymiar = 10;
boolean[][] komorki = new boolean[wymiar][wymiar];
```

[12] źródło: http://www.openprocessing.org/browse/?viewBy=tags&tag=game%20of%20life (dostęp: 1.03.2014)

Powyższy kod tworzy tablicę o nazwie „komorki” o wymiarach 10 na 10 pozwalając przechowywać w poszczególnych komórkach wartości 0 lub 1. Następnie konieczne jest wypełnienie komórek losowymi wartościami na początek (stan wyjściowy gry). Robimy to za pomocą zagnieżdżonych pętli for:

```
for(int rzad=0; rzad<wymiar; rzad++) {
     for(int kolumna=0; kolumna<wymiar; kolumna++) { //losuj[13] }}
```

Pętla for przechodzi przez wszystkie rzędy naszej tablicy (od 0, sprawdzając czy jest w ostatnim rzędzie [czyli rząd<wymiar], jeśli nie dodając +1 do numeru rzędu [++ oznacza „zwiększ o 1”]) i analogicznie przez wszystkie kolumny. Wewnątrz pętli zamieszczamy kod wypełniający komórki przez które przechodzimy losowymi wartościami binarnymi:

```
komorki[kolumna][rząd]=losujWartosc();
```

Dzięki tym operacjom uzyskujemy tablicę o wymiarach 10 na 10 (czyli składającą się ze 100 pól) wypełnioną wartościami, które będą nas informowały o tym czy poszczególne pole jest „żywe” lub „martwe”. Pozostanie jedynie sprawdzenie wartości poszczególnej komórki za formuły umieszczonej w identycznej jak powyżej pętli for i wykonanie na niej jakiejś operacji wizualizacyjnej np. pomalowania zawartości komórki na czarno (wartość koloru „0”) lub biało (wartość koloru „255”). Zastosujemy do tego warunek if:

```
if(komórki[kolumna][rząd]) fill(0); else fill (255);
```

Możemy zwrócić uwagę na fakt, że powyższy warunek if nie ma sformułowanego explicite pytania („jeśli co”) – dzieje się tak dlatego, że komórki mają już przypisaną wartość boolowską, którą można interpretować jako 0 lub 1, „tak” lub „nie” albo „warunek zachodzi” i „warunek nie zachodzi”. W związku z tym uniwersalna odpowiedź na nasz if jest już przechowywana w komórce tablicy. Teraz wystarczy, dla każdej komórki (znów podwójna pętla for) sprawdzimy czy każdy sąsiad jest „żywy” czy „martwy”. Każda komórka ma potencjalnie ośmiu żywych sąsiadów (z wyjątkiem tych, które znajdują się na obrzeżach układu np. narożne będą miały maksymalnie trzech, pozostali będą niejako automatycznie

[13] w języku Processing „//” oznacza komentarz czyli tekst nie kompilowany do wykonywanego programu, tutaj komentarz oznacza miejsce gdzie znaleźć się powinna procedura losująca zawartość komórki.

„martwi" ponieważ ich otoczenie jest z dwóch stron puste). Po zliczeniu liczby sąsiadów każdej komórki możemy zaimplementować wypisane we wstępie tego podrozdziału reguły gry i rozstrzygnąć czy przeżyje ona, obumrze lub w przypadku gdy jest martwa i ma dokładnie trzech żywych sąsiadów – ożyje. Dalsze szczegóły implementacji programu pozostają mniej istotne dla naszych rozważań w tym miejscu – interesowało nas bowiem jedynie zastosowanie tablic do wdrożenia reguł *Gry w Życie* w naszym systemie. Kod źródłowy działającego programu wraz z komentarzami znajduje się w załączniku na końcu dysertacji[14].

Na powyższym przykładzie widać w jaki sposób zastosowanie konkretnych struktur danych jakimi są arraye służy do stworzenia programu będącego artystycznym wykorzystaniem wizualizacji pewnego konceptu matematycznego.

Przypadek drugi: modelowanie dynamicznych struktur narracyjnych za pomocą GRAFÓW (narzędzia digital humanities).

Jednym z najbardziej interesujących narzędzi humanistyki cyfrowej *(digital humanities)* są programy i skrypty umożliwiające analizę różnego rodzaju obiektów kultury w aspektach, które były dotąd dla klasycznego aparatu humanistycznego niedostępne lub bardzo trudne do ujęcia ze względu na swoją ogromną skalę. Rozwijaniem tego rodzaju softwaru zajmuje się między innymi prowadzone przez Lwa Manovicha – znanego badacza nowych mediów – laboratorium Software Studies Initiative. Pod jego egidą powstał na przykład program do wizualizacji ImagePlot[15]. Bardzo często do badania relacji pomiędzy różnego rodzaju znaczącymi (w sensie: wyposażonymi w metadane) obiektami stosuje się struktury grafowe, które wyśmienicie nadają się do modelowania zależności pomiędzy różnymi zjawiskami i obiektami.

Prostym przykładem narzędzia wykorzystującego modelowanie grafowe jest propozycja stworzenia holistycznego modelu dynamicznych struktur narracyjnych opracowana przez badaczy z Wydziału Fizyki, Astronomii i Informatyki Stosowanej UJ w artykule pt. *Towards Graph-Based Model of Computer Games* (Bartosz Porębski, 2012). Należy zaznaczyć, że propozycja ma charakter przyczynkowy i nie ujmuje należycie złożoności zjawisk towarzyszących – mówiąc językiem Espena Aarsetha – ergodycznej

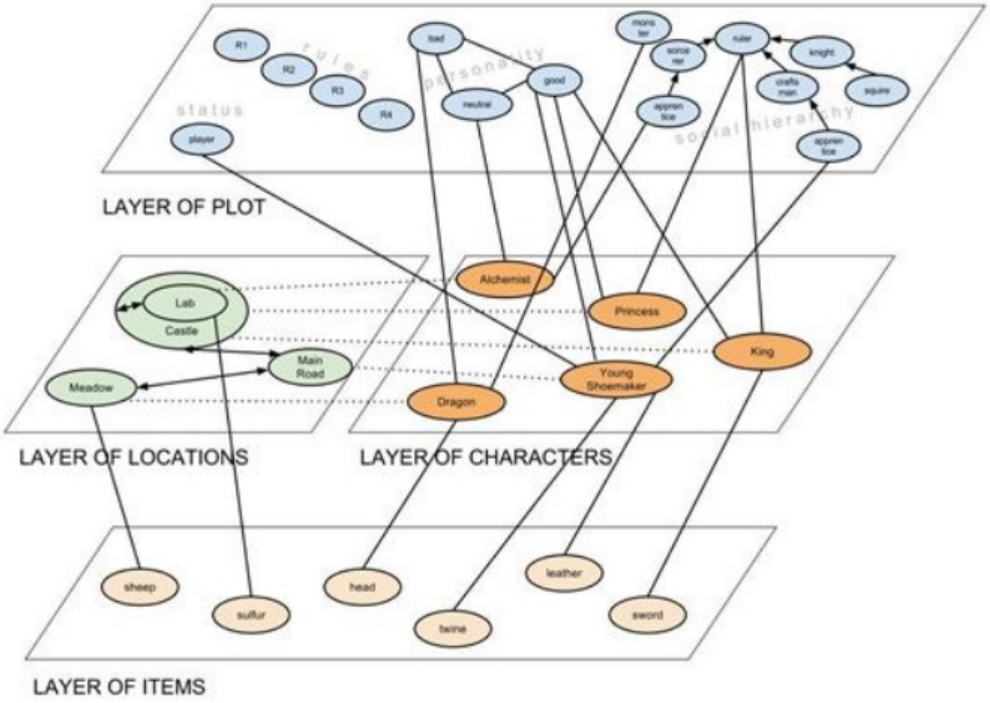
---

[14] vide: „*Z 1: Kod źródłowy programu Gra w Życie"*

[15] http://lab.softwarestudies.com/p/imageplot.html (dostęp: 1.03.2014)

lekturze cybertekstu, nie mniej jednak interesująco pokazuje przydatność narzędzia jakim jest modelowanie za pomocą grafów.

Koncept polega na wydzieleniu z materiału jakim jest gra wideo klas obiektów takich jak „postaci", „lokacje", „przedmioty", „elementy fabularne" oraz odpowiednim zaprezentowaniu relacji zachodzących między nimi oraz ich zmian.



Ilustracja 13: Opis przykładowego świata i fabuły dokonany za pomocą grafów (za: Bartosz Porębski, 2012)

Ujęcie takie pozwala na przeprowadzenie formalizacji w celu umożliwienia sprawdzania spójności świata, wyszukiwania fabularnych „ślepych uliczek". Gracz pokonując kolejne przeciwności realizuje produkcje poszczególnych grafów, których możliwa liczba jest duża lecz skończona. Dodatkowo poszczególne węzły opisywanego systemu posortowane są w warstwy, z których każda posiada pewną indywidualną charakterystykę i w pewien sposób określa relacje w jakie element znajdujący się na węźle może wejść z elementami znajdującymi się na innych warstwach.

Reasumując - bez przytaczania konkretnych wyjaśnień matematycznych – zastosowanie specyficznego rodzaju struktury informacyjnej (danych) jaką jest graf zyskujemy możliwość szczegółowego opisu zachowań dynamicznego systemu.

A 2.2. Algorytmy.

Jak zwraca uwagę David Harel we wstępie do swojej książki pt. Rzecz o istocie informatyki (Harel, 2000) algorytmika to coś więcej niż tylko dział nauki o komputerach. To dziedzina wiedzy, która leży i podstaw wszelkiego cyfrowego przetwarzania informacji. Jej znaczenie jest fundamentalne. Historia algorytmu sięga – jak już wiemy – czasów znacznie wcześniejszych niż wynalazek elektronicznej maszyny liczącej. Nie mniej jednak to dopiero za sprawą nowoczesnych układów elektronicznych, procesorów – jego rola stała się tak istotna dla współczesnego krajobrazu naszej cywilizacji.

Ponieważ bardzo podstawowa definicja algorytmu była już wyżej przytaczana, skupimy się na poszerzeniu zagadnień związanych z działaniem tej kategorii zjawisk oraz ich wpływem na kulturę, sztukę i społeczeństwo. Punktem wyjścia dla naszych rozważań będzie koncepcja zadania algorytmicznego i jego rozwiązań przeprowadzanych za pomocą abstrakcyjnych przepisów opisujących kolejne działania jakie mogą zostać podjęte w celu jego możliwie efektywnego rozwiązania. Następnie przyjrzymy się jak aktywności te mogą być realizowane za pomocą specyficznych struktur danych oraz konkretnych czynności opisywanych w języku (tu: językach programowania).

Zadanie algorytmiczne to taka konstrukcja, która zbudowana jest z:

„(1) scharakteryzowania dopuszczalnego, być może nieskończonego zbioru potencjalnych zestawów danych wejściowych oraz
(2) specyfikacji pożądanych wyników jako funkcji wyjściowych”

(Harel, 2000, p. 30)

Oznacza to, że podaje nie tylko informacje na temat tego jak rozwiązać dany problem ale też jakiej postaci wyniku należy się spodziewać. Powoduje to, że możliwa jest automatyzacja pozyskanych w ten sposób wyników (np. przekazanie ich do innego algorytmu, podprogramu etc.). Z poprzedniego rozdziału wiemy, że dla procesora bardzo ważna jest kolejność wykonywania operacji, jesteśmy świadomi też, że liczba konkretnych wykonywanych przez jednostkę obliczeniową operacji jest bardzo ograniczona. Dlatego istotną dla algorytmu kategorią są instrukcje elementarne – ich opis musi zostać zrealizowany w taki sposób, żeby dał się jednoznacznie przełożyć na pojedyncze działania

wykonywane w specyficznej kolejności. Stąd zbiory instrukcji sterujących, które dają się przełożyć na konkretne „zachowania” procesora.

W cytowanej powyżej publikacji Harel wyróżnia struktury przepływu sterowania w postaci:

- bezpośredniego następstwa (wykonaj czynność A, a następnie czynność B)
- wyboru (rozgałęzienia) warunkowego (jeśli Q to A, w przeciwnym wypadku B)

Dają się one realizować w iteracjach tj. pętlach z określoną ilością wykonań (zrób A dokładnie N razy) lub (znów) warunkowych (zrób A do momentu Q). Przykład tego rodzaju instrukcji widzieliśmy w programie konstruującym tablice na potrzeby *Gry w Życie.* Polecenie losujące było wykonywane tyle razy, ile wynosiła zadana liczba komórek (arbitralna liczba, nazwijmy ją „k”) w ten sposób, że po każdej iteracji (wypełnieniu komórki wartością binarną) program dodawał do licznika wartość 1 (w kodzie licznik++) i porównywał z liczbą N (komórek w rzędzie oraz kolumn). Jeśli licznik<N operacja była wykonywana ponownie. Ponieważ w programie mieliśmy do czynienia z zagnieżdżeniem pętli w drugiej takiej samej strukturze – operacja była wykonywana dla każdego rzędu w każdej komórce (w sumie k czyli NxN razy). Zwraca to uwagę na fakt, że dzięki konstrukcjom iteracyjnym istnieje możliwość opisania bardzo krótkimi poleceniami bardzo nieraz złożonych procesów. Dobrym przykładem tego rodzaju operacji jest przebieg algorytmu tzw. sortowania bąbelkowego. Polega on na tym, że porównywane są dwa kolejne elementy układu i zamieniana jest ich (i tylko ich) kolejność w zależności od zadanego czynnika. Proces jest wykonywany iteracyjnie a jego realizacja kończy się kiedy następuje przejście całego zbioru bez żadnej zamiany kolejności. Kod źródłowy przykładowego programu sortującego znajduje się w załączniku na końcu dysertacji[16].

Bardzo ważną instrukcją sterującą jest też polecenie skoku (skocz do G). Polega ona na tym, że w pewnym miejscu algorytmu (np. spełnieniu jednego z warunków) program wykonuje przejście do innego, wskazanego miejsca. Jego użycie ułatwia realizowanie pewnych zadań (w szczególności obsługę sytuacji wyjątkowych) jednak wpływa negatywnie na przejrzystość całej struktury – pomijanie części algorytmu może też w pewnych przypadkach prowadzić do zaburzenia pracy programu. Nie mniej jednak jest to operacja dopuszczalna i w wielu przypadkach stosowana.

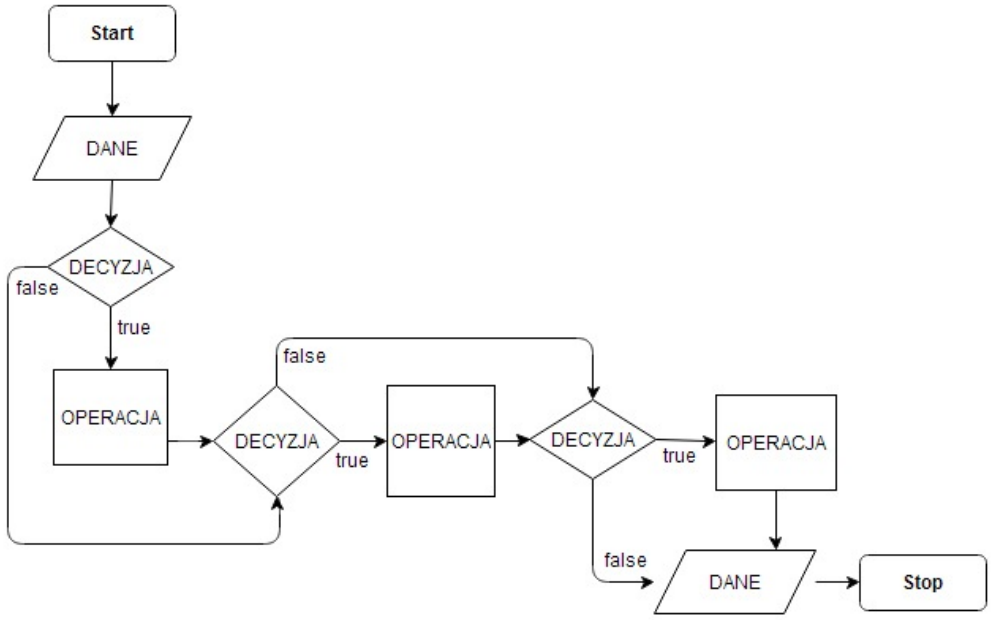
---

[16] vide: „*Z 2: Kod źródłowy programu Sortowanie Bąbelkowe (Processing)”*

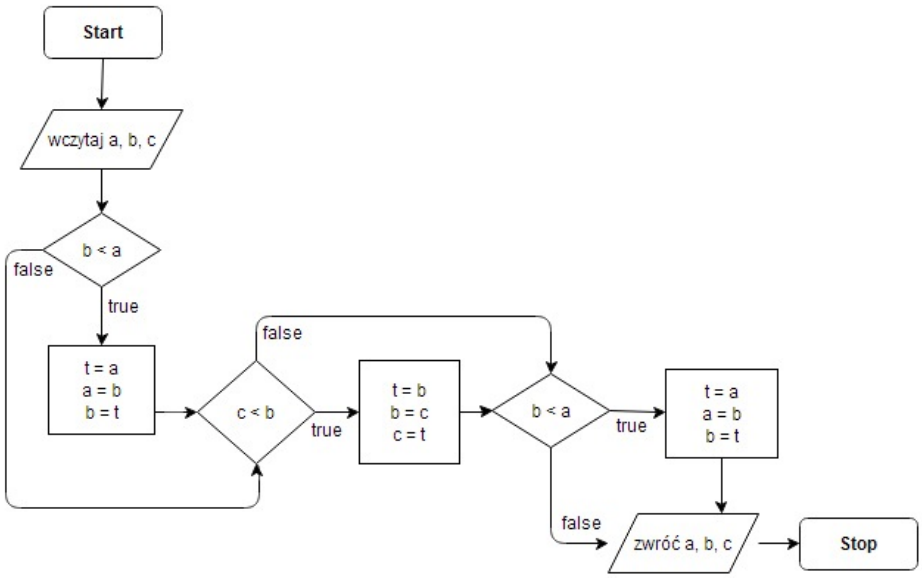
A 2.2.1 Schematy blokowe algorytmów. Procedury i rekurencja.

Dla wygody projektantów algorytmów stworzony został metajęzyk, który umożliwia ich opisywanie w formie diagramów (*flowchart*) ułatwiających szybkie rozpoznanie sposobu ich działania. Poszczególne elementy grafiki przedstawiają procedury, systemy, decyzje, operacje i inne elementy niezbędne do działania programu czy wykonania czynności lub rozwiązania problemu. Należą do nich blok startowy, symbol pobierania/zapisu danych, element oznaczający konieczność podjęcia decyzji (najczęściej prawda/fałsz dla zadanego warunku), wykonanie operacji, blok kończący wykonanie zadania.

Każdy z wymienionych elementów oraz łączące je relacje zostały uwidocznione na ilustracji poniżej zaś konkretny podprogram (pojedynczy przebieg operacji sortowania) opisany za pomocą diagramu przedstawia obraz następny.



Ilustracja 14: Typowe funkcje elementów w schemacie blokowym.



ilustracja 15: Schemat blokowy algorytmu sortującego.

Skoro już przywołane zostało pojęcie podprogramu (procedury/*subroutine*) powinniśmy zdefiniować je jako zestaw poleceń służący do zrealizowania pojedynczego zadania ujęty jako „autonomiczna" jednostka. „Samodzielność" podprogramu wyraża się tym, że może on być wywoływany w różnych miejscach kodu i parametryzowany w zależności od potrzeb. Możemy np. napisać procedurę wyszukującą szereg znaków w tekście i wywołać ją w jednym miejscu do odnalezienia słowa „kot" a w innym ciągu „pies". Bardzo często konkretne podprogramy wraz z instrukcjami ich użycia pojawiają się jako biblioteki stosowane w różnych językach programowania. Podprogramy, w zależności od okoliczności mogą przybierać postać funkcji (mają wykonać obliczenia i zwrócić pewną wartość), procedur (wykonują pewne działania w rodzaju wyświetlenia obiektu czy wyczyszczenia ekranu) czy metod (w programowaniu obiektowym to element klasy, który ma za zadanie wykonanie dla niej operacji[17]). Ogólna, choć rzadko używana nazwa wszystkich podprogramów to „jednostka wywoływana". W Processingu możemy spotkać tego rodzaju konstrukcje we wszystkich wymienionych wariantach. Najprostszym przykładem jest operacja losowania zawartości komórki w naszym programie *Gra w Życie* – jest ona wykonywana w podwójnej pętli czyli dla każdej komórki tablicy osobno za każdym razem realizując jednak to samo zadanie:

[17] więcej informacji o klasach, obiektach, i metodach znajduje się w rozdziale *„A4.2 Paradygmat obiektowy"*

```
boolean losujKomorki() {
      float a = random(100);
      if(a>50) return true; return false; }
```

w powyższym kodzie `losujKomorki()` będzie miało wartość zerojedynkową (boolean) uzyskaną w wyniku wylosowania losowej liczby w zakresie do 100 („a") a następnie zwrócenia 1 jeśli wylosowana liczba będzie większa niż 50 lub 0 jeśli będzie mniejsza. Taka procedura jest nam potrzebna ponieważ w Processingu nie ma polecenia pozwalającego automatycznie wylosować liczbę binarną – musimy więc sobie taką operację skonstruować ze „zwykłego" polecenia `random();` odnoszącego się do liczb a nie wartości true/false.

Zwrócenie uwagi na fakt, że pewna funkcjonalna, zamknięta całość może być częścią programu i być „podpinana" w różnych jego punktach wiedzie nas ku pojęciu rekurencji (rekursji). Jak pisze Harel „jest to zdolność podprogramu (procedury) do wywołania samego siebie" (Harel, 2000, p. 46) i podaje przykład rozwiązania problemu znanego jako Wieże Hanoi (różnej wielkości obręcze nanizane na słupki należy przenieść po jednym z pierwszego słupka na ostatni w ten sposób, że większy nigdy nie może znaleźć się nad mniejszym). Jeśli A, B, C to kołki, zaś N to krążki – zadanie można rozwiązać za pomocą procedury rekurencyjnej:

```
     procedura przenieś N z X(A) na Y(B) używając Z(C);
(1) jeśli N = 1, to wypisz „X → Y";
(2) w przeciwnym razie (tj. jeśli N>1) wykonaj co następuje:
     (2.1) wywołaj przenieś N – 1 z X (A) na Z(C) używając Y(B);
     (2.2) wypisz „X → Y"; (2.3) wywołaj przenieś N – 1z Z(C) na
     Y(B) używając X(A);
(3) wróć
```

(za: Harel, 2000, p. 46)

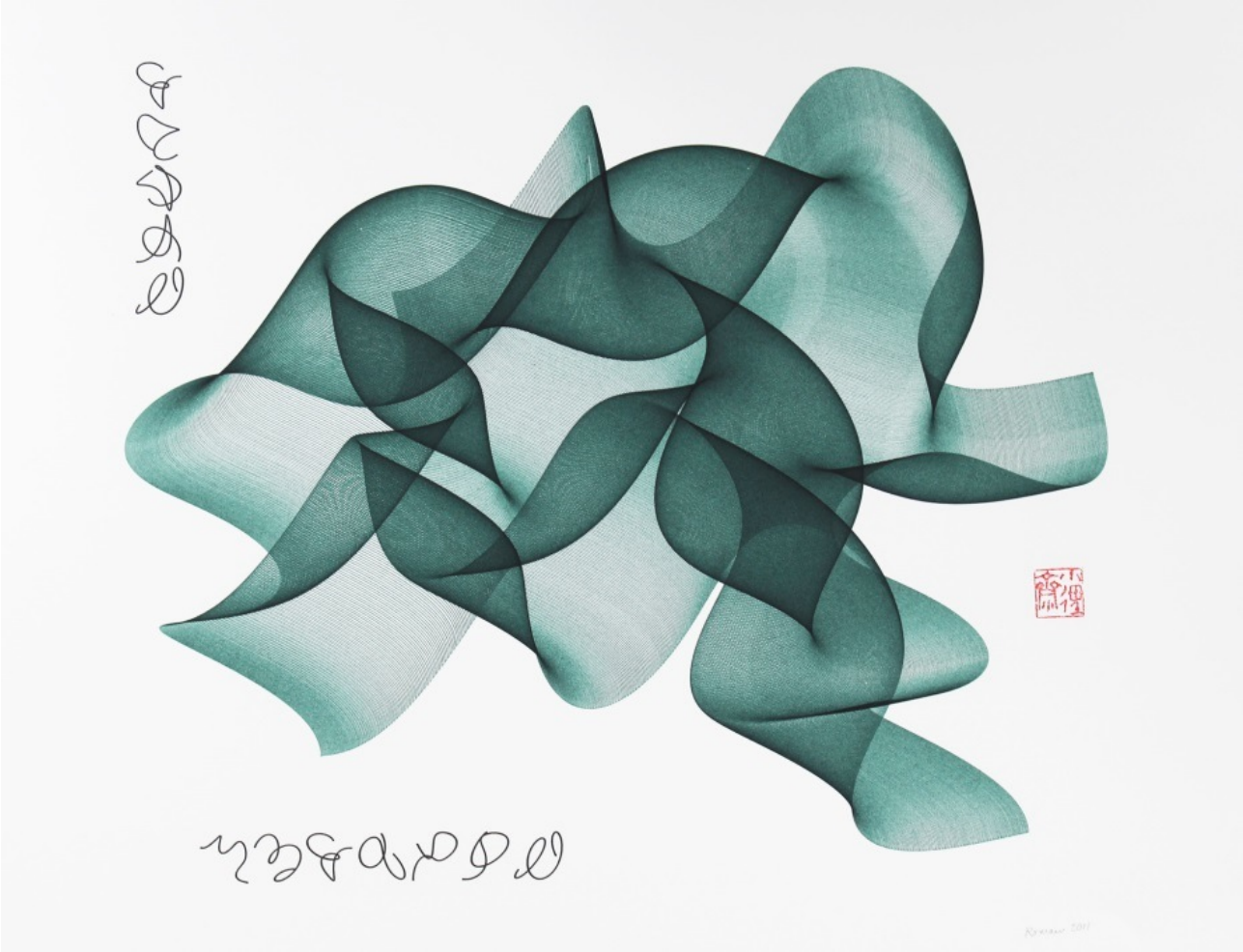
A 2.2.2 Sztuka algorytmiczna / sztuka generatywna.

Algoryści

Jak wynika z powyższych rozważań algorytm jest specyficznego rodzaju strukturą logiczną, która może zostać zaimplementowana w kodzie źródłowym programu komputerowego. Szczególnie zwraca się uwagę na konstruowanie algorytmów, które w sposób optymalny będą mogły służyć realizowaniu pragmatycznych zadań związanych z obliczeniami, sortowaniem, przetwarzaniem danych. Dlatego tak istotnym parametrem jest złożoność obliczeniowa algorytmu oraz kwestia tego, czy zadane operacje wykonuje on tak wydajnie jak to tylko możliwe. Jednak algorytmika służyć może nie tylko konstruowaniu kodu o wysokich walorach użytkowych. Do dziedzin ludzkiej aktywności, w których znajduje ona zastosowanie należą również obszary związane z działalnością stricte artystyczną. Można oczywiście powiedzieć, że wszelkiego rodzaju aktywność twórcza, która odbywa się za pośrednictwem mediów cyfrowych lub tworzona jest przy użyciu programów komputerowych należy w jakiś sposób do „sztuki algorytmów" ponieważ z całą pewnością takowe – przynajmniej pośrednio, a bywa że i świadomie – aplikuje. Przedmiotem naszego zainteresowanie są jednak ci autorzy, którzy z algorytmów uczynili sobie podstawowe narzędzie i cel pracy artystycznej.

Jedną z wczesnych, choć całkowicie nieformalnych grup identyfikujących swą działalność bezpośrednio z nowym medium byli twórcy, którzy sami nazwali się później Algorystami. Słowo to oznacza generalnie każdego, kto zajmuje się algorytmami – ma więc ono szerokie, historyczne, sięgające czasów przed komputerowych znaczenie. Jednak - jak pisze jeden ze współtwórców ruchu (choć „ruch" to być może nieco za mocne określenie) Roman Verostko – Algoryści zaadoptowali pojęcie, czyniąc je znakiem osób zajmujących się konkretnie sztuką algorytmiczną. Początki działalności twórców identyfikujących się jako the Algorists, czyli m.in. Jean Pierre Heberta, Manfreda Mohra, Romana Verostko i Marka Wilsona sięgają zarania komputerów przeznaczonych do użytku indywidualnego, czyli okolic roku 1970. Algoryści przyjęli swoją nazwę za sprawą Heberta niejako retrospektywnie w 1995. Twórczość artystów koncentruje się do dziś wokół generatywnych możliwości urządzeń komputerowych. Ich idee związane z twórczą ekspresją pozostają w ścisłej korelacji z prezentowaną wcześniej w niniejszej pracy refleksją Chomsky'ego na temat generatywnego charakteru pewnych struktur logicznych. Maszyna umożliwiająca implementację jego teorii stała się też automatycznie wehikułem

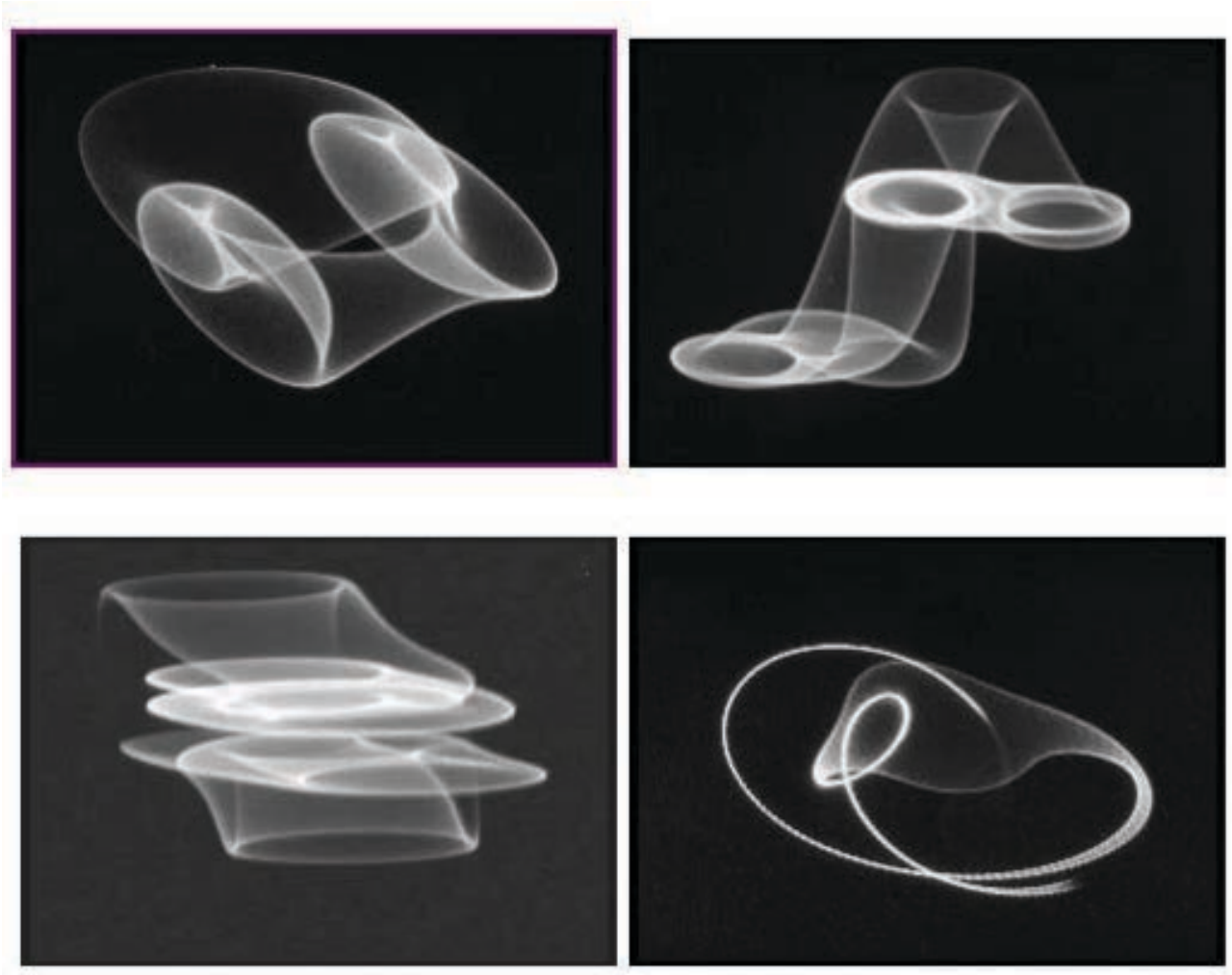
dla nowych - generatywnych właśnie - form estetycznych. Jak pisze Verostko[18] – wyrastając ze wspólnego korzenia sztuki geometrycznej i konceptualnej XX wieku Algoryzm wyforsował się na mającą źródło w technologii autonomię.



Ilustracja 16: Roman Verostko, *Green Cloud (the drawing),* pióro i tusz na ploterze, 2011

Autor *Zielonej chmury* zwraca także uwagę na fakt, że algorytmiczne ukorzenienie uprawianej przezeń i jego przyjaciół sztuki nie powinno być mylone z odwoływaniem się do „czystej" matematyki. Matematyczność dzieł wynika z oczywistego osadzenia w procedurach najlepiej opisywalnych za pomocą języka Królowej Nauk. Nie mniej jednak ich wyjątkowość wyraża się w fakcie implementacji w konkretnym elektronicznym narzędziu za pomocą zapisu w języku programowania. To czyni sztukę algorytmiczną interesującą z naszego punktu widzenia. Oto bowiem komputerowe oprogramowanie staje się samodzielnym środkiem artystycznego wyrazu.

[18] Strona internetowa autora: http://www.verostko.com/algorist.html (dostęp: 1.05.2014)



Ilustracja 17: Herbert Franke, *Electronic Graphics,* 1961-62

Język programowania jest na tyle elastycznym środkiem ekspresji, że możliwe jest nawet przeprowadzenie definicji „Algorysty” za pomocą pseudokodu. Jean Pierre Hebert sformułował ją następująco:

```
if
(creation && object of art && algorithm && one's own algorithm)
     {
     include * an algorist *
     }
elseif
(!creation || !object of art || !algorithm || ! one's own algorithm)
     {
     exclude * not an algorist *
     }
```
[19]

Ilustracja 18: A. Michael Noll, *Computer composition with lines,* 1964

Grupa identyfikująca wspólnym mianem osób pracujących z algorytmicznym medium to oczywiście jeden z wielu przykładów twórców zaangażowanych w tworzenie sztuki w oparciu o koncepcje związane z podstawami informatyki i działania oprogramowania w ogóle. Dziś sztuka generatywna pozostaje ważnym elementem nowoczesnej działalności artystycznej.

Sztuka generowana proceduralnie

Pod pojęciem sztuki generowanej proceduralnie (lub po prostu – proceduralnej) rozumiemy takie formy estetycznej wypowiedzi, które realizowane są za pomocą systemów autonomicznych. Chodzi o sytuację, w której narzędzie jest przez artystę w jakiś sposób programowane (nie musi tu jednak chodzić o kodowanie w sensie informatycznym) do wykonywania dzieła. Rolą twórcy jest więc sformułowanie pewnych

[19] Korespondencja Hebert-Verostko datowana na 31 sierpnia 1995 (za stroną internetową autora)

warunków wyjściowych oraz ustawienie (*setup*) systemu – sama realizacja pozostaje już poza jego bezpośrednią kontrolą. Sztuka proceduralna zakłada pewien element nieokreśloności jeśli chodzi o efekt końcowy dzieła. Chodzi mianowicie o to, żeby zaprogramowany obiekt wytworzył strukturę, która jest przewidywalna tylko w pewnym stopniu. Istotnym elementem pracy staje się kwestia losowości, swobodnego lub organicznego powstawania różnorodnych tworów – stąd właśnie odwołanie do pojęcia „generatywności". Elementy przetwarzania proceduralnego możemy napotkać na gruncie praktycznie wszystkich ważnych sztuk:

- w muzyce dawniejszej (J. P. Kirnberger) i współczesnej (John Cage, Brian Eno)
- w sztukach wizualnych (Sol LeVitt, Woody Vasulka, J.-P. Hebert i Algoryści, Mark Napier)
- w literaturze (Borroughs, Tristan Tzara)

Sztuka generatywna pomimo, że nie odnosi się wyłącznie do dziedziny opartej na programowalnych układach elektronicznych (jest kategorią szerszą) – pozostaje w ścisłej relacji w stosunku do sztuki algorytmicznej (znów, kategoria – teoretycznie – szersza niż sztuka komputerowa) oraz *software artu*. Za cezurę wyznaczającą wejście tego rodzaju twórczości na estetyczne salony świata najlepiej prawdopodobnie uznać wystawę Cybernetic Serendipity, która została zapoczątkowana w 1968 roku w londyńskim Institute of Contemporary Arts. Dziś generatywność jako formuła realizowania różnorodnych przedsięwzięć, również tych o charakterze artystycznym nie wydaje się niczym szczególnie niezwykłym.

Przykład: „Sea and Spar Beetween" Nicka Montforta i Stephanie Strickland

Omawiane dzieło jest generatorem poezji, który realizuje różne kombinacje strof i wyrażeń w przestrzeni dwuwymiarowej (o koordynatach 0,0 na 14992383, 14992383) co daje – jak piszą autorzy[20] - około 225 trylionów możliwych kombinacji. Generowanie wierszy odbywa się za pomocą poruszania kursorem po wyznaczonym obszarze lub po wpisaniu ręcznie dowolnych koordynatów (z podanego zakresu) w pole tekstowe. Strofy i wrażenia pochodzą z wierszy Emily Dickinson oraz z *Moby Dicka* Hermana Melville i są

[20] http://blogs.saic.edu/dearnavigator/winter2010/nick-montfort-stephanie-strickland-sea-and-spar-between/ (dostęp: 1.05.2014).

zestawiane dzięki zaimplementowaniu pewnych struktur generatywnych w kodzie źródłowym programu.

Sam program ma postać strony internetowej – typowego dokumentu HTML5 z elementem canvas, który pozawala na wyświetlanie wyniku dzięki użyciu biblioteki canvastext.js oraz formularzem pobierającym położenie kursora. Silnikiem programu jest wykonywany przez przeglądarkę kod JavaScript (zamieszczony w załączniku na końcu dysertacji wraz z komentarzami). W skrypcie znajdują się zmienne odpowiedzialne za przechowywanie i mieszanie wyrażeniami charakterystycznymi dla danego autora (`varDickinsonSyllable`, `varMelvilleSyllable`) a także siedem możliwych linijek (np. `varShortPhrase`, `function ExclaimLine`, `function riseAndGoLine`).

Ilustracja 19: „See and Spar Between” w akcji.

Warto przy okazji wspomnieć, że w 2013 roku zostało zrealizowane polskie tłumaczenie poematu dokonane przez Monikę Górską-Olesińską i Mariusza Pisarskiego, którego wynik wraz z analizą został zaprezentowany podczas konferencji Hold The Light 2014 Electronic Literature Organization Conference[21].

A 2.2.3 Algorytmy w *digital humanities.*

David M. Berry jeden z ciekawszych badaczy i jednocześnie krytyków humanistyki cyfrowej w swoim tekście *The Computational Turn* zwraca uwagę na pewien fundamentalny fakt związany z nowym paradygmatem badań w tej dziedzinie nauk:

„Cyfrowa humanistyka próbuje oddać sprawiedliwość formom digitalnym i temu jak zmieniają one sposoby naszej pracy z reprezentacją i mediacją. Możemy to nazwać cyfrowym »składaniem« rzeczywistości dzięki któremu mamy dostęp do kultury w radykalnie nowy sposób. Aby operować medialnie obiektem, urządzenie cyfrowe czy komputerowe wymaga aby obiekt ten został przetłumaczony na digitalny kod, który może ono zrozumieć."

(Berry, 2011a, p. 1)

Metodologia nauk humanistycznych w coraz większym stopniu wspiera się na różnego rodzaju narzędziach cyfrowych. Szczególnie przydają się one w przypadku badań prowadzonych nad fenomenami wymagającymi modelowania ze względu na swoją złożoność oraz trudnymi do ujęcia ze względu na dużą skalę ilościową. W takich zadaniach techniki przetwarzania komputerowego, wspomagane operowaniem na metadanych sprawdzają się wyśmienicie. Pracując jednak z różnego rodzaju oprogramowaniem do algorytmicznego przetwarzania danych kulturowych musimy cały czas pamiętać o kluczowych kwestiach wyeksponowanych w pierwszej części tej pracy: informacje cyfrowe mają swoją specyficzną charakterystykę wynikającą przede wszystkim z tego, że ich podstawą jest kwantyfikacja tj. dzielenie rzeczywistości na dyskretne fragmenty dające się opisać numerycznie. Jest to o tyle istotne, że w procesie konwersji z „analogowego" na „cyfrowe" pominięta musi zostać pewna „przestrzeń pomiędzy" poszczególnymi skwantyfikowanymi fragmentami. Skala ujmowania zjawisk (rozdzielczość) jest ściśle uzależniona od technologii, których używamy do cyfryzowania

[21] Treść prezentacji dostępna pod adresem: http://conference.eliterature.org/sites/default/files/papers/olesinska_pisarski_SSB_final_414_1.pdf (dostęp 1.05.2014).

obiektów rzeczywistości. Fakt ten może być pomijalny w niektórych wypadkach, np. jeśli digitalizujemy tekst i opisujemy jako dane poszczególne litery, to nie możemy zrobić tego bardziej szczegółowo niż z dokładnością do pojedynczej litery – co więcej nie mamy takiej potrzeby. A więc tekst utworu literackiego nie traci na „jakości" po jego zamianie na dane – przetworzeniu ulega tylko sposób dostępu do niego – oraz, co fundamentalne, zyskujemy możliwość jego komputerowej analizy. Zysk jest więc jednoznaczny. Inaczej wygląda sytuacja w przypadku obiektów wizualnych (np. zdjęć analogowych) czy zapisów muzycznych. Kwantyfikacja dokonuje się z pewną arbitralną, zadaną rozdzielczością – oczywiście możemy proces ten przeprowadzić z ogromną dokładnością, taką która z całą pewnością nie może być percypowana przez nasz aparat poznawczy (jedynym skutkiem ujemnym jest to, że większa dokładność oznacza większe pliki i większe zużycie przestrzeni na nośniku). Nie mniej jednak – pewna redukcja tutaj niewątpliwie zachodzi. Filozoficzne pytanie, które można zadać: czy skanując utwór muzyczny nie odzieramy go z pewnej istotnej wartości, na jakimś, być może na razie nie do końca opisanym poziomie? Wszak wartość instrumentów Antonio Stradivariego wynika z niesłychanej, nieuchwytnej, subtelnej doskonałości ich brzmienia. Jest to niewątpliwie trudna do rozstrzygnięcia kontrowersja, której konkretne rozwiązania muszą odbywać się w odniesieniu do konkretnych przypadków pracy z materiałem medialnym. Tak czy inaczej faktem pozostaje zwiększająca się wokół nas liczba obiektów o charakterze digitalnym i rosnąca potrzeba zajęcia się nimi w sposób, który pozwoli na ujęcie i wykorzystanie ich unikatowych właściwości.

W tym rozdziale naszym podstawowym zadaniem jest poszerzenie pojęcia algorytmu oraz wskazanie tego jak wchodzi on w zakres zainteresowania *digital humanities* przez fakt bycia podstawowym elementem działającego softwaru. Szersza refleksja na temat charakteru cyfrowej humanistyki i implikowanych przez nią narzędzi badawczych znajduje się w podsumowaniu części Δ niniejszej pracy pt. *Interfejs między częściami 2. Od Δ do T: Digital Humanities. Oprogramowanie kultury i hermeneutyka informatyki* oraz w rozdziale *T 1 Humanistyka cyfrowa i zwrot komputacyjny.*

Przykład: Computational Stylistics Group

Trzech badaczy związanych z Uniwersytetem Jagiellońskim, którzy tworzą Computational Stylistics Group to Maciej Eder (Uniwersytet Pedagogiczny w Krakowie, Polska Akademia Nauk), Jan Rybicki (Uniwersytet Jagielloński), Mike Kestemont (Uniwersytet w Antwerpii). Stworzyli oni pakiety o nazwie „stylo" dla środowiska *R* – jednego z najbardziej znanych i rozpowszechnionych na świecie programów do przetwarzania statystycznego i graficznego[22]. Rozwiązania zaproponowane przez naukowców - a szczególnie ich uniwersalność, spowodowały że należą one do najpopularniejszych spośród aktualnie używanych tego rodzaju pakietów w Europie i na świecie.

Stylometria jest bardzo interesującą dziedziną wiedzy, która polega na wszechstronnej analizie zróżnicowanych tekstów pod wybranymi przez badaczy aspektami. Przydatność tego rodzaju zabiegów zależy od założonej tezy i badanego materiału. Dzięki badaniom stylometrycznym możliwa jest np. pomoc w rozstrzyganiu kontrowersji dotyczących autorstwa niektórych dzieł (analizie poddawano m.in. utwory Homera czy Szekspira), a także badanie sieci podobieństw stylistycznych między różnymi korpusami (ze względu autora, epokę, rejon geograficzny).

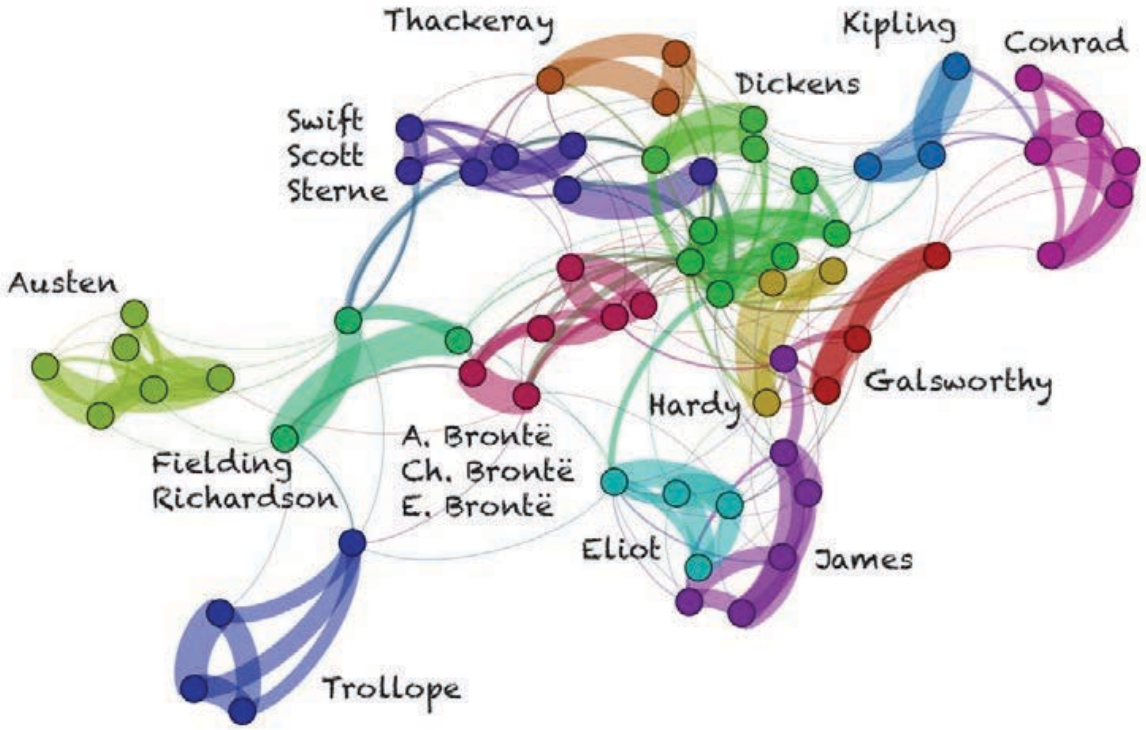
Sam pomysł badań stylometrycznych nie jest nowy. Za pierwszą tego rodzaju analizę uważa się przeprowadzony w 1439 roku przez Lorenzo Vallę dowód na fałszerstwo *Donacji Konstantyna*. Udało się tego dokonać dzięki porównaniu języka jakim zapisany był dokument z innymi – bez wątpienia pochodzącymi z epoki. W 1890 roku Wincenty Lutosławski zaprezentował metodę, którą wykorzystał do badań nad dialogami Platona – polegała ona na ilościowym zestawieniu rozkładu stylemów (słów, wyrażeń, kolokacji).

Dzisiaj stylometria może rozwijać się dzięki rozwojowi komputerów oraz algorytmiki. Okazuje się bowiem, że można dokonywać badań nad całymi, liczącymi dziesiątki tysięcy stron, korpusami (lub pakietami) dzieł – rzecz trudna lub wręcz niemożliwa do wykonania dla pojedynczego badacza pracującego metodami nie korzystającymi ze wsparcia informatycznego.

---

[22] Strona projektu: http://r-project.org/ (dostęp 1.05.2014)

Oczywiście uzyskane wyniki zależą od założonej tezy oraz od konstrukcji algorytmów badających teksty. Stąd narzędzia stylometryczne jawią się jako doskonałe uzupełnienie niektórych badań humanistycznych lecz oczywiście nie mogą w żaden sposób zastąpić rozumnej refleksji wykształconego naukowca. Wnioski z ustalonych relacji i ich sensowność zależą całkowicie od przyjętego planu badawczego.



Ilustracja 19: Stylistyczni „najbliżsi sąsiedzi" przedstawieni w formie sieci połączeń[23]

[23] Źródło ilustracji: https//sites.google.com/site/computationalstylistics/projects/testing-consensus-networks (dostęp 1.05.2014)

Przykład: ImagePlot, oprogramowanie Software Studies Initiative (NEH[24], Calit2[25], CRCA[26])

ImagePlot jest projektem realizowanym przez Software Studies Initiative, międzynarodową grupę badawczą, której centralną siedzibą są media laby Calit2 (California Institute for Telecomunication and Information Theory) i od niedawna – po przenosinach pomysłodawcy i szefa inicjatywy Lwa Manovicha - CUNY (City University of New York).

Oprogramowanie to wpisuje się w dwa najbardziej modne trendy związane z cyfrową humanistyką – analizę dużych zbiorów informacji (*Big Data*) oraz nowoczesne techniki wizualizacyjne, które mają mieć charakter wyjaśniający a nie tylko być obiektami estetycznymi (vide: Tufte, 2006). Jak zwraca uwagę Manovich w dzisiejszych czasach ogarnięcie zjawiska treści generowanych przez użytkowników (*user generated content*) staje się drastycznie coraz trudniejsze – od „nowych mediów" przemieściliśmy się w stronę „więcej mediów" (Manovich, 2013). W jeden dzień na portalu Flikr pojawia się więcej zdjęć niż istnieje obrazów we wszystkich galeriach sztuki na świecie. Aby podjąć próbę choćby częściowego prześledzenia zjawisk mających miejsce w ramach nowej, porażającej ilością wytworów, cyfrowej kultury naukowcy rozpoczęli pracę nad czymś co sami nazywają nowym paradygmatem w badaniu jej artefaktów, dynamik i przepływów – Cultural Analytics. Zadanie tyleż ambitne, co trudne do wykonania. Nie mniej jednak samo założenie, że użycie wielkoskalowej analizy komputerowej i interaktywnej wizualizacji wzorców kulturowych pomoże w rozumieniu niektórych fenomenów dynamicznej nowej kultury wydaje co najmniej interesujące.

Pierwszym wyzwaniem jakie stanęło przed twórcami Cultural Analytics było skonstruowanie oprogramowania, które pozwoli na analizę danych – w pierwszej kolejności tych o charakterze wizualnym (wideo, animacje, filmy, gry komputerowe, zdjęcia, grafika) – tak powstał ImagePlot. Oprogramowanie pozwala na wczytanie danych do bazy, a następnie ich analizę oraz przetworzenie na chmurę punktów, z których każdy zawiera szereg informacji na temat pojedynczego obiektu (obrazu, klatki filmu). Możemy manipulować wizualizacjami na różne sposoby, operując metadanymi i kiedy doszukamy

[24] National Endowment for Humanities (NEH)

[25] The California Institute for Telecommunications and Information Technology (Calit2)

[26] The Center for Research in Computing and the Arts (CRCA)

się interesującego naszym zdaniem wzorca (np. w relacji miedzy chronologią i saturacją barwną klatek w twórczości jakiegoś reżysera) – wyrenderować wysokiej jakości grafikę lub interaktywną, nawigowalną mapę, która w zamyśle ma pozwolić badaczowi przyjrzeć się konkretnym obiektom i pomóc w znalezieniu odpowiedzi na pytanie o naturę badanej zależności. Zadziwiająca jest pewność z jaką zespół Manovicha formułuje wnioski z tego rodzaju prezentacji obrobionych komputerowo danych. ImagePlot i Cultural Analitycs niosą ze sobą aurę narzędzia opartego o algorytmikę, więc matematycznego – to znaczy obiektywnego. W przypadku podobnych projektów łatwo jest ulec pozorowi naukowości, takiej jak rozumieją ją *sciences*. Czujnemu badaczowi kultury, medioznawcy w obliczu tak prezentowanych rezultatów (z pewnością jaką czyni to Manovich) powinno się „zapalić światło ostrzegawcze". Wszak jednym z głównych założeń studiów nad oprogramowaniem jest stwierdzenie, że software nie pozostaje „niewinny" w swoim użyciu, a obiektywność algorytmów kończy się na poziomie ich analizy matematycznej – ich wykorzystanie otwiera zupełnie osobną kategorię zjawisk i interpretacji. Stąd projekty związane z ilościowymi analizami fenomenów kultury spotykają się z coraz częstszymi krytykami ze strony co bardziej wrażliwych badaczy – należą do nich np. wspomniany wcześniej David Berry czy Aleksander Galloway.

Ilustracja: „Chmura" przedstawiająca okładki magazynu Time
pod względem ostrości (oś X) i saturacji barwnej (oś Y)

Sposoby pracy z oprogramowaniem stylometrycznym oraz z programem ImagePlot zostały opisane się w rozdziale: *T 2 Media Lab Toolkit. Kod, dokumentacja, przypadki użycia* pod koniec dysertacji.

A 2.2.4 Znaczenie algorytmu.

Jak to zostało przed chwilą wykazane algorytmy są zjawiskami, które pomimo swojej ulotnej, wysoce abstrakcyjnej charakterystyki przenikają całość softwarowego uniwersum stanowiąc logikę i mechanikę jego działania. Maszyny Turinga, komputery, są jedynie urządzeniami do przetwarzania danych. W powyższych rozważaniach zaprezentowana została natura ich działania, która okazała się być czysto analityczna. Tym, co czyni pracę urządzeń elektronicznych znaczącą dla nas jest kwestia doboru danych, które przetwarzają oraz - co chyba istotniejsze - tego *jak* są one procesowanie. Wraz z uruchomieniem tego rodzaju pytań otwarte zostaje całe uniwersum nowych problematyzacji sposobów na jakie działa algorytm - a co za tym idzie oprogramowanie. Proste i podstawowe operacje logiczne zyskują swoją nową głębię kiedy umieścimy je w kontekstach związanych z pytaniem o autora (*kto* przetwarza te dane?), motywację (*po co* te dane są przetwarzane?) i metodę (*dlaczego* te dane przetwarzane są *tak właśnie*, a nie inaczej?). Prawdziwym znaczeniem algorytmu nie jest jego sens immanentny - jako struktury matematyczno-logicznej lecz to na jakie procesy społeczne, ekonomiczne, estetyczne on wpływa. Oczywiście, musimy sobie zdawać cały czas sprawę z tego z jak złożonymi zjawiskami obcujemy - sam algorytm jako taki nie wpływa bowiem bezpośrednio na nic, działa dopiero opracowane na jego bazie oprogramowanie i to tylko wtedy gdy uruchomione zostanie w określonym kontekście i zasilone pewnymi danymi. Nie mniej jednak wydaje się, że da się wyróżnić pewne obszary, w których wpływ *cybermedium* okazuje się najistotniejszy ze względu na dobór algorytmu bardziej niż innych warstw działania komputerowego softwaru. Jedną z propozycji takiego wyróżnienia jest lista zaproponowana przez Tarletona Gillespie w artykule *The Relevance of Algorithms* (Traleton Gillespie, 2014, p. 179). Wymienia on sześć wymiarów tego jak algorytmy mogą posiadać społeczną (polityczną) "wartościowość". Odbywa się to przez:

1. Wzorce włączania *(patterns of inclusion),*
2. Cykle oczekiwań *(cycles of anicipation),*
3. Ewaluacja trafności *(the evaluation of relevance),*
4. Obietnica algorytmicznej obiektywności *(the promise of algorithmic objectivity),*
5. Powiązanie z praktyką *(entangelement with practice),*
6. Produkcja "wykalkulowanego" odbiorcy *(the production of calculated publics).*

Ad 1. /patterns of inclusion/

Nie ulega wątpliwości, że "algorytmy są bezwładnymi, pozbawionymi znaczeń maszynami dopóki nie zostaną połączone z bazami danych na których mają funkcjonować" (ibid.). Autor zwraca uwagę na fakt, że z punktu widzenia odbiorcy/ użytkownika poszczególne warstwy działania komputerowego aparatu medialnego nie są rozróżnialne (co istotne - rzadko kiedy mogą takimi być). W związku z *przeźroczystością* współdziałania algorytmu z bazą danych istnieje tu potężne pole do różnego rodzaju manipulacji i to w bardzo dogodnym, bo sprawiającym wrażenie obiektywnego ze względu na swój numeryczno-logiczny charakter środowisku. Gdybyśmy chcieli przyjrzeć się sposobom konstruowania wzorców włączania i wykluczania moglibyśmy za Gillespie wprowadzić podział ze względu na:

- *Zbieranie danych.* Na szczególną uwagę zasługują dyskursy problematyzujące to co publiczne/prywatne, otwarte/zamknięte, darmowe/komercyjne. Bardzo ważna w tym kontekście jest oczywiście hegemoniczna pozycja korporacji Google, ukonstytuowana przecież dzięki opracowaniu innowacyjnego silnika wyszukującego (i indeksującego!) informacje. Działa on na podstawie (dziś już niesłychanie rozbudowanego i przekształconego) algorytmu o nazwie PageRank, który został opracowany przez założycieli Google: Larrego Page'a i Sergeya Brina, pod koniec lat dziewięćdziesiątych na Stanford University. Kwestiom wyszukiwania i indeksacji przyjrzymy się jeszcze w dalszej części pracy - na razie wystarczy zwrócenie uwagi na fakt, że inwazyjność (potrafi indeksować dane pozornie lub za słabo zabezpieczone przed przeglądaniem) i bezsporna dominacja silnika Google w znaczącym stopniu definiuje zakres tego, co dostępne dla popularnego użytkownika Internetu a co nie.

- *Preparację pod algorytm.* Wiemy z poprzednich rozdziałów, że w uniwersum informacyjnym nie istnieje praktycznie coś takiego jak "surowe dane" (a jeśli już to raczej jako obiekt postulowany ontologicznie a nie coś z czego korzysta teoria informacji), zawsze bowiem posiadają one jakąś ustanowioną strukturę. Mogą one oczywiście być z perspektywy przetwarzania lepiej lub gorzej zorganizowane lecz ich uformowanie stanowi jeden z głównych warunków możliwości traktowania występującej w świecie *diafory* jako obiektu o charakterze informacyjnym. Sposoby formowania danych i protokoły dostępu do nich są neutralne dla tego jak działa informacja np. różne ontologie baz danych owocują odmiennymi interakcjami użytkownika z ich zawartością.

- *Wykluczenie.* Z metodologii kulturoznawstwa wiemy, że z punktu widzenia krytycznej analizy równie istotne jak to co dany dyskurs priorytetyzuje jest to co pomija lub programowo wyłącza poza swój nawias. Gillespie zwraca uwagę, że np. takie portale jak YouTube (własność Google) algorytmicznie obniżają ranking kontrowersyjnych filmów, tak aby nie pojawiały się one na listach najchętniej oglądanych klipów czy ekranach propozycji dla nowych użytkowników. Podobnie Twitter, który generalnie nie cenzuruje wpisów (poza oczywiście drastycznymi przypadkami) lecz blokuje te, które algorytm uznał za kontrowersyjne/wulgarne z rankingu trendów.

Ad 2. /cycles of anticipation/

W przypadku mediów poddanych porządkowi algorytmu obowiązują zwykle mechanizmy znane z klasycznych mediów masowych lecz bardzo często w wyolbrzymionej postaci. Tak jest również w przypadku obserwowanej swoistej samozwrotności komunikacji. Wykorzystanie algorytmów jako podstawy w regulowaniu zindywidualizowanego dostępu do informacji polega zwykle na wykorzystaniu danych na temat aktywności użytkownika (dynamicznej - w czasie rzeczywistym lub - częściej - zgromadzonych w historii jego interakcji z programem). Napędzany tego rodzaju wiedzą silnik zwraca wyniki związane siecią poprzednich rezultatów swojej pracy. Tworzy to swoistą spiralę informacyjną. Oczywiście aktywność użytkownika nie zawęża się jedynie do korzystania z danych dostarczanych przez silnik na bazie jego uprzedniej aktywności - osoby korzystające np. z wyszukiwarek internetowych czy systemów polecania filmów lub muzyki mają zawsze możliwość wprowadzenia niepowiązanych, "spontanicznych" informacji wpływając na poszerzenie sieci powiązań. Nie mniej jednak Sieć zbiera wiedzę na nasz temat korzystając z różnego rodzaju ekosystemów informacyjnych (np. Facebook "wie" skąd się łączymy [aplikacja mobilna], kogo tagujemy, co lubimy, gdzie jadamy [połączenie np. z Foursquare], co oglądamy [połączenie z Netflix], jakiej muzyki słuchamy [połączenie z iTunes, Last.fm, Spotify], w jakie gry gramy [wtyczka PlayStation Network, Steam, Apple App Store, Google Play] etc. etc.) i wykorzystuje ją do skonstruowania naszej *algorytmicznej tożsamości* (Cheney-Lippold, 2011). Poza wszelkiego rodzaju korzyściami jakie przynosi ten model dystrybucji dostępu do wiedzy, ma on również swoją mroczną stronę jaką jest mniej lub bardziej zindywidualizowane profilowanie odbiorców treści ze względu na status społeczny, możliwości konsumpcyjne etc. Coraz więcej mówi się na temat wykorzystania tego rodzaju danych przez banki, pracodawców czy nawet służby wywiadowcze państw lub policję. Temu tematowi również jeszcze się przyjrzymy.

Istotne, że jeśli chcemy korzystać z dobrodziejstw Internetu - mamy stosunkowo niewielkie pole do manewru i małą ilość możliwości stawiania czynnego oporu w stosunku do inwigilujących nas mechanizmów Sieci.

Ad 3. /the evaluation of relevance/

Algorytm jest mechanizmem, który wcielony w kod i uruchomiony na pewnym zbiorze danych zwróci pewien wynik. Uniwersalność procedury jest tu kwestią kluczową. Abstrakcyjne mechanizmy tworzy się po to, aby uzyskiwać relewantne wyniki dla różnych zbiorów danych. W przypadku przetwarzania różnego rodzaju danych statystycznych czy medycznych istnieje model za pomocą którego jesteśmy w stanie sprawdzić czy przetwarzanie danych doprowadziło do satysfakcjonującego rezultatu. Nie istnieje natomiast obiektywna metryka sprawdzająca trafność wyników wyszukiwania, informacja na ten temat musi zostać dostarczona przez użytkownika, który również nie wypowiada się przecież w sposób obiektywny lecz może jedynie ocenić swoją satysfakcję z osiągniętego rezultatu. Włączanie takiego kryterium *(user experience, UX)* do mechaniki z ducha matematycznego systemu stanowi kolejny interesujący aspekt działania współczesnych algorytmów.

Ad 4. /the promise of algorithmic objectivity/

Użycie algorytmu stanowi w powszechnym przekonaniu swego rodzaju "gwarant" zaufania. Programom komputerowym przypisuje się specyficzną *cybernetyczną podmiotowość* z takimi przymiotami jak obiektywność, dokładność, bezstronność, bezbłędność. Warto zwrócić uwagę, że w przypadku kontrowersji związanych z algorytmicznymi silnikami najczęściej oskarża się ich twórców o "manipulowanie algorytmem" (np. regularne tezy na temat ręcznego sterowania PageRankiem przez Google czy EdgeRankiem przez Facebooka) a nie o "preparowanie" go. Procedura jawi się więc jako coś obiektywnego i sprawdzalnego, a nie twór, który może być po prostu przystosowany do partykularnych celów swojego autora.

Gillespie zwraca uwagę, że celebrowanie *algorytmicznej obiektywności* przypomina podkreślanie roli bezstronności i rzetelności jako normy stanowiącej dla zachodniego dziennikarstwa:

"Podobnie jak silniki wyszukiwarek, dziennikarze wypracowali taktyki sprawdzania co jest najbardziej trafne, jak to zaprezentować, jak zapewnić tego relewantność - zbiór praktyk, które są relatywnie niewidzialne dla odbiorcy, a co do których wiadomo, że są one znacznie mniej eleganckie niż by się mogło wydawać (...) Dziennikarze używają norm obiektywności jako »strategicznego rytuału« aby pozyskać legitymację społeczną dla taktyk produkcji wiedzy, które są z natury niepewne"

(Traleton Gillespie, 2014, p. 181)

To bardzo interesujące spojrzenie zwraca naszą uwagę na fakt, że nowe media algorytmiczne pozostają w uwikłane w "stare dobre" dyskursy medialne w daleko większym zakresie niż skłonni byliby przyznać ich użytkownicy, twórcy a nawet niektórzy badacze.

Ad 5. /entangelement with practice/

Kolejną właściwością indukującą *znaczenie algorytmu* jest jego związanie z pragmatyką. W ogromnej większości przypadków są one konstruowane w celu efektywnego zrealizowania jakiejś czynności. Przypadki tworzenia algorytmów celowo wadliwych i z zamierzenia nieefektywnych należą do ekstremalnej rzadkości. Praktyczny wymiar konstrukcji algorytmu jest oczywiście najbardziej widoczny w oprogramowaniu, które służy za bezpośrednie narzędzie do interakcji ze środowiskiem funkcjonowania człowieka - od zbiorów reguł regulujących pracę klimatyzacji w upalny dzień po autonomiczne oprogramowanie reagujące w milisekundowych opóźnieniach na zmiany kursu giełdy walutowej. W takim przypadku lepiej jest mówić nie o *wpływie* algorytmu na użytkownika technologii lecz o wzajemnym *powiązaniu* aktorów. "Niezwykle istotnym jest abyśmy postrzegali to splątanie nie jako jednokierunkowy wpływ lecz jako rekursywną pętlę między obliczeniami algorytmu a »kalkulacjami« ludzi" - pisze Gillespie (ibid.).

Ad 6. /the production of calculated publics/

Algorytmy są w równym stopniu kluczowym, technologicznym komponentem zmediatyzowanych środowisk, co stanowią podłoże z którego się one wyłaniają. Nie tylko pozwalają na przekształcanie użytkowniczych taktyk i praktyk lecz również strukturyzują nasze interakcje z pozostałymi członkami usieciowanej społeczności. Jak zwraca uwagę autor komentowanego tekstu: algorytmy sterują ruchem między "wykalkulowanymi"

odbiorcami. Pod pojęciem tym kryje się obliczeniowa z natury praktyka tworzenia zupełnie nowych sieci komunikacyjnych na bazie danych wskazanych programowi jako istotne. Kiedy np. sklep internetowy poleca nam przedmioty, które kupili klienci zainteresowani tymi samymi produktami co my - stworzona zostaje właśnie taka relacja. Facebook dysponuje różnorodnymi trybami widoczności postu - od prywatnego (tylko ja) do publicznego (wszyscy). Jednak kiedy ustawiamy, że informacja może być widoczna przez "naszych przyjaciół oraz przyjaciół naszych przyjaciół", generujemy nowe struktury grafu społecznościowego, które działają na podstawie algorytmu "transformującego zamknięty zbiór użytkowników w publiczność" (ibid.). Stąd różnica między siecią odbiorców stworzoną przez użytkownika a "wykalkulowaną" przez algorytm publicznością naszej aktywności. Oczywiście obie struktury nakładają się na siebie czyniąc relacje relacje społecznościowe na tyle złożonymi, że aż wychodzącymi poza możliwości percepcyjne pojedynczego odbiorcy. Stąd podziwiana niejednokrotnie przez nienawykłych odbiorców *magia* Facebooka czy Naszej Klasy (w jej najlepszym okresie), które to portale *potrafią* w zaskakujący nieraz sposób "odkopać" znajomych czy wydarzenia, o których dawno zapomnieliśmy.

Powyższe przykłady prezentują w jaki sposób możemy rozumieć kwestie *znaczenia algorytmu.* Chodzi mianowicie nie tyle o jego własną semantykę, której jako zjawisko z pogranicza matematycznej abstrakcji jest raczej pozbawiony, lecz o to jak będące jego kwintesencją *działanie* rekonfiguruje to co znaczące w otaczającej nas rzeczywistości (głównie komunikacyjnej, aczkolwiek nie tylko). Podane przykłady koncentrowały się nieco nadmiernie na tematyce związanej z silnikami wyszukiwarek internetowych oraz różnego rodzaju usługami społecznościowymi - stało się tak dlatego, że po pierwsze: w obiektach tego rodzaju najłatwiej zaobserwować procesualność (jesteśmy *czynnymi* uczestnikami działania algorytmów, a nie jedynie *odbiorcami* efektów ich działania); po drugie: w tym momencie naszych rozważań brakuje nam jeszcze pojęć i struktur aby mówić stricte o oprogramowaniu. Tą zaległość należy więc jak najszybciej nadrobić, nadmieniając jednak, że do tematyki związanej z sieciowością, relacjami, protokołami i kontrolą dystrybucji wiedzy i informacji wrócimy jeszcze niejednokrotnie.

A 2.3 Programy / oprogramowanie (*software*).

W prowadzonych rozważaniach coraz częściej, na przemian z pojęciem algorytmu pojawia się kwestia *oprogramowania*. Są one w oczywisty sposób powiązane, jednak w żadnym przypadku nie możemy mówić o ich tożsamości. Zanim zajmiemy się szerzej omówieniem problematyki związanej z programowaniem (czyli praktyką tworzenia *softwaru*) wypada - po raz kolejny - dokonać systematyzacji podstawowych pojęć.

Zadziwiające, że wśród znacznej już dziś wielości prac poświęconych studiom nad oprogramowaniem, a także w nieprzeliczonym bogactwie publikacji na temat tworzenia oprogramowania z trudem możemy doszukać się precyzyjnej definicji tego czym właściwie ono jest. Możemy dowiedzieć się, że "programy stanowią w końcu skonkretyzowane sformułowania abstrakcyjnych algorytmów na podstawie określonej reprezentacji i struktury danych" (Writh, 2000, p. 11) czy "oprogramowanie jest duszą i ciałem naszej maszyny (Turinga - dop. JKA) (...) stanowi z grubsza wszystko, co komputer może zrobić" (Floridi, 1999, p. 47). Z pomocą w systematyce przychodzi więc nieoceniona Wikipedia, źródło nie zawsze przez wszystkich doceniane jednakże w przypadku artykułów z dziedziny IT wyjątkowo rzetelne:

"Oprogramowanie - czytamy - to całość informacji w postaci:
- zestawu instrukcji,
- zaimplementowanych interfejsów,
- zintegrowanych danych

przeznaczonych dla komputera do realizacji wyznaczonych celów.

Celem oprogramowania jest przetwarzanie danych w określonym przez twórcę zakresie.
Oprogramowanie jest synonimem terminów program komputerowy oraz aplikacja (...)
Oprogramowanie tworzą programiście w procesie programowania (...)
Oprogramowanie pisane jest zazwyczaj przy użyciu różnych języków programowania z wykorzystaniem algorytmów.
Programy przekształcające oprogramowanie z postaci źródłowej na binarną to kompilatory."[27]

[27] http://pl.wikipedia.org/wiki/Oprogramowanie (dostęp 1.06.2014)

I dalej:

"Oprogramowanie dzieli się ze względu na przeznaczenie:
- oprogramowanie systemowe - realizujące funkcje konieczne dla działania systemu komputerowego,
- oprogramowanie do tworzenia oprogramowania,
- biblioteki programistyczne - oprogramowanie do wykorzystania przez inne programy,
- oprogramowanie użytkowe - mające bezpośredni kontakt z użytkownikiem i realizujące usługi dla tego użytkownika za pomocą aplikacji, z technicznego punktu widzenia jest to oprogramowanie korzystające z usług m.in. systemu operacyjnego, a szerzej - oprogramowania systemowego."[28]

W kolejnych rozdziałach pracy przyglądać się będziemy temu jak oprogramowanie jest tworzone - co (ze względu na paradygmaty [vide: A4] oraz różne języki programowania [vide: A5]) bezpośrednio determinuje sposób jego funkcjonowania.

A 2.3.1 Oprogramowanie jako sztuka. Kilka słów o demoscenie.

Oprogramowanie zasadniczo służy pragmatyce przetwarzania danych. Z punktu widzenia użytkownika najczęściej do czynienia mamy - zgodnie z wyliczeniem z poprzedniego podrozdziału - z oprogramowaniem systemowym lub narzędziowym (użytkowym, w tym rozrywkowym). Nikogo nie dziwi zatem stosowanie programów komputerowych do tworzenia dzieł o charakterze artystycznym. Aplikacje takie jak pakiety graficzne Adobe, software do montażu cyfrowego materiału filmowego czy tworzenia muzyki są w powszechnym użyciu wśród twórców sztuki. Interesująco (z naszego punktu widzenia) zaczyna się robić, kiedy ktoś zaczyna postrzegać samą działalność programistyczną w kategoriach artystycznych. W poprzednich rozdziałach zaprezentowana została twórczość Algorystów, którzy swoją twórczość utożsamiali z konstruowaniem grafiki na bazie struktur geometryczno-algorytmicznych. Z podobnymi inspiracjami mamy do czynienia w przypadku demosceny. Stanowi ona jednak zjawisko o tyle interesujące, że powstało całkowicie oddolnie, samorzutnie i spontanicznie - bez

---

[28] tamże

jednolitego manifestu czy fundującej ruch wystawy. Z dala od tzw. *świata sztuki*[29], przy okazji zupełnie innego rodzaju aktywności.

Co to jest demoscena?

Zacznijmy jednak od początku. Sformułowanie odpowiedzi na pytanie o to, czym właściwie jest scena demo jest znacznie trudniejsze niż wykazanie jej pochodzenia. W ramach samego ruchu również nie ma pełnej jasności w tej kwestii - szczególnie jeśli chodzi o taktowanie fenomenu w kategoriach artystycznych. Ujmowanie prac scenowców jako sztuki budziło, szczególnie w przeszłości, stanowczy opór niektórych spośród nich. Wynikało to z jednej strony z silnie anarchizującego podłoża, na którym wyrosła demoscena, z drugiej zaś z faktu, że większość twórców identyfikowała się jako "komputerowcy" czyli pasjonaci technologii, cyberpunki, programiści, elektronicy - ich postawa wobec *świata sztuki* była co najwyżej obojętna, często zaś negatywna. Spowodowało to, że demoscena wyrosła w środowisku podziemnym, *undergroundzie* - z dala od jakichkolwiek wystaw, mód estetycznych czy analiz teoretycznych. Ruch, który w czasach swojej największej świetności jednoczył dziesiątki tysięcy ludzi z całego świata spotykających się na potężnych *parties,* w zasadzie nie funkcjonował w dyskursie głównego nurtu. Dziś, przede wszystkim dzięki powszechnej obecności komputerów i komunikacji internetowej stan rzeczy uległ nieco zmianie. Zainteresowanie retro komputerami, archeologią tego medium, moda związana z platformami Atari, Commodore czy Amiga musiała doprowadzić do szerszego włączenia demosceny w "świat zewnętrzny". Wszak jest ona pełna ludzi, którzy "starych komputerów" nie przestali nigdy używać - mało tego, doprowadzili ich obsługę do mistrzostwa i znaleźli sposoby na łącznie ich z nowoczesnym sprzętem (np. z płaskimi ekranami wysokiej rozdzielczości czy współczesnymi nośnikami danych, takimi jak karty SD). Dzisiejsza demoscena dorosła wraz z ludźmi, którzy się na niej wychowali. A bardzo wielu spośród nich stanowi współczesną elitę świata IT (wszak zajmowali się hackowaniem sprzętu i programowaniem w czasach, kiedy większość ich rówieśników dopiero zaczęła mieć do czynienia z pierwszymi w swoim życiu komputerami).

[29] Posługuję się pojęciem "świata sztuki" technicznie, w rozumieniu instytucjonalnym jako zbioru obejmującego muzea, galerie, festiwale, publikacje "branżowe", teoretyków, historyków, krytyków i wreszcie artystów "akredytowanych" przez wyżej wymienionych, zdając sobie sprawę ze złożoności pojęcia "artworldu" w rozumieniu np. Artura Danto. W tym miejscu chciałbym jednak uniknąć uwikłania w tego rodzaju problematykę teoretyczno-artystyczną.

Na jednej z głównych stron internetowych - agregatorów ruchu demoscenowego (demoscene.org) czytamy:

"Demo jest rezultatem współpracy wielu młodych programistów, grafików i muzyków. Pracują oni jako grupa *(demogroup)* nad programami demonstracyjnymi *(demo),* w których prezentują swoje umiejętności w programowaniu graficznym i algorytmicznym, generowanej komputerowo grafice i muzyce. Tymi demami grupy mogą konkurować z innymi na dużych imprezach *(parties)* w zróżnicowanych konkursach (*competition, compo*) na całym świecie"[30]

z artykułu *about* na stronie jednej z najciekawszych, współczesnych demogrup - *Conspiracy,* dowiedzieć się zaś możemy, że:

"Demoscena, mówiąc po prostu jest ogólnoświatową, niekomercyjną subkulturą sztuki komputerowej, która specjalizuje się w tworzeniu dem"[31]

okazuje się więc, że mówienie o demoscenie w kategoriach sztuki jest dziś jak najbardziej na miejscu, choć w dalszym ciągu nie znajdziemy na światowej mapie wydarzeń *electronic artu* zbyt wielu przedsięwzięć związanych z tą tematyką.

Z perspektywy rozważań będących osią niniejszej dysertacji najbardziej frapujący wydaje się jednak początek i rozkwit sceny komputerowej - a więc spojrzenie historyczne. Pozwoli ono prześledzić jak medium obliczeniowe (a w szczególności samo oprogramowanie) może stać się źródłem i centrum specyficznego dyskursu (jednak) artystycznego, który z maszyn cyfrowych uczynił sobie nie tyle kolejne narzędzie i środek ekspresji lecz zbudował wokół nich całą swoją tożsamość.

[30] http://www.demoscene.org/demoscene.php (dostęp 1.07.2014)

[31] http://conspiracy.hu/about/demoscene/ (dostęp 1.07.2014)

Grafika komputerowa.

Historia demosceny to w dużej mierze dzieje rozwoju technologii komputerowych. Kiedy urządzenia elektroniczne wyszły z okresu swojego niemowlęctwa i zostały wyposażone w graficzne interfejsy oraz peryferia, na których dało się *wyświetlać* wyniki ich pracy - zaczęły pojawiać się pierwsze pomysły na ich wykorzystanie w duchu, który możemy dziś nazwać demoscenowym. Proto-scena sięga czasów Prawdziwych Programistów (vide rozdział A 3.1) i ich zmagań z konstrukcjami lampowymi. Do pionierów specyficznego sposobu myślenia o należeli np. tacy twórcy jak John Whitney, który w 1958 roku użył pokładowego urządzenia celowniczego samolotu *B-29 Superfortress* do generowania geometrycznych wizualizacji użytych m.in. przez Hitchcocka w czołówce filmu *Vertigo* (*Zawrót Głowy*).

Ilustracja 19: Wizualizacja Whitneya w filmie *Vertigo* Hitchcocka (1958)

Poza eksperymentami z grafiką komputerową realizowaną w celach innych niż czysto narzędziowa pragmatyka, we wczesnych latach sześćdziesiątych pojawiła się również tendencja do wykorzystania (wielkich jeszcze i przede wszystkim drogich) komputerów w celach rozrywkowych. Legendarny twórca pierwszej gry komputerowej William "Harry" Higinbotham ze swoim *Tennis for Two* uruchomił lawinę zjawisk, które zaowocowały między innymi dalszym "kreatywnym nadużywaniem" sprzętu badawczego, szczególnie przez studentów dużych uczelni technicznych (takich jak MIT). Zanim udało się zmniejszyć rozmiary urządzeń komputerowych i zredukować ich cenę na tyle by mogły trafić pod dachy szeregowych konsumentów - co było podstawowym warunkiem narodzin i rozwoju *właściwej* demosceny, w świecie grafiki komputerowej doszło do jeszcze kilku istotnych wydarzeń. Należy do nich m.in. rozwiązanie zagadki specyficznej budowy algorytmu modelującego wybrzeże Wysp Brytyjskich przez matematyka, szefa IBM

Thomas J. Watson Research Center - Benoit B. Mandelbrota. Badania poskutkowały stworzeniem niezwykłych, dających się w piękny sposób wizualizować struktur matematycznych, zwanych fraktalami. Ich wyjątkowość polega, najprościej rzecz ujmując na tym, że możemy *powiększyć* dowolną część z nich i uzyskamy w ten sposób kolejną iterację fantazyjnych wzorów, których krawędzie znów możemy *powiększyć* i tak w nieskończoność (a właściwie do momentu wyczerpania mocy obliczeniowej procesora dokonującego "powiększeń", które tak naprawdę są obliczeniami). Publikacja badacza z 1978 roku (*The Fractal Geometry of Nature)* oraz późniejsze poprawki innych naukowców (np. Carpentera, inżyniera Boeinga) doprowadziły do szerokiego zastosowania fraktali w CGI (*Computer Generated Imaginery).* Szczególnie nadawały się one do generowania krajobrazów oraz teksturowania płaszczyzn (użyto ich np. w 1977 do stworzenia ścian Gwiazdy Śmierci w filmie *Star Wars* Lucasa).

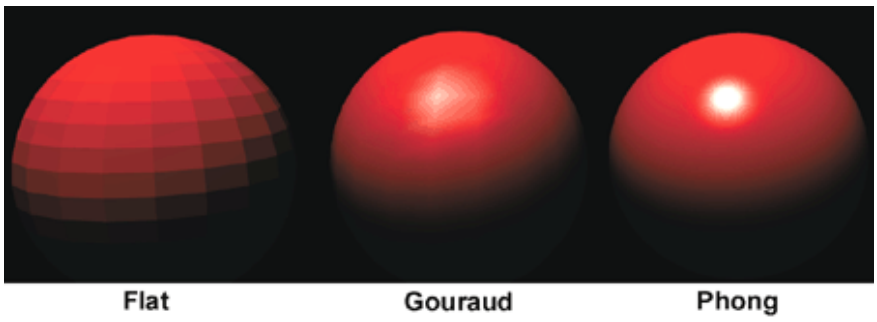
Ilustracja 20: Fraktalnie wygenerowany krajobraz z dema *Elevated* grupy *RGBA* (2009)[32]

Do innych interesujących odkryć należało również - omawiane już wcześniej - opracowanie reguł *Gry w Życie* przez Hortona Conwaya z Uniwersytetu Cambridge.

Rozwój grafiki komputerowej to przede wszystkim aplikowanie złożonych koncepcji matematycznych i geometrycznych, a także fizyki do pracy z nowym rodzajem hardwaru.

[32] Ciekawostka: kod dema, w którym przez ponad trzy minuty "podróżujemy" po meandrach górskiego krajobrazu ma 4 kilobajty... włączając to ścieżkę muzyczną.

Już w latach sześćdziesiątych opracowane zostały pierwsze urządzenia, które potrafiły operować grafiką wektorową w przestrzeni trójwymiarowej (stary LDS-1 zastąpiony DAC-1 IBM-a i General Motors, komputer przeznaczony do pracy z CAD-ami, czyli programami wspierającymi pracę projektantów i inżynierów [*CAD - Computer Aided Design, DAC - Design Augmented by Computers])*. Grafika wektorowa, ze względu na swój matematyczny charakter pozostaje do dzisiaj jednym z najważniejszych narzędzi grafiki komputerowej, szczególnie na potrzeby pracy z realizacjami dla wsparcia różnego rodzaju wytwórstwa (druk wielkoformatowy, inżynierski CAD etc.). Dużą rewolucją było jednak pojawienie się komputera wyposażonego w ekran rastrowy (wyświetlający piksele) jakim był IBM 2250. Dalszy rozwój grafiki 3D to przede wszystkim (fundamentalne!) pojawienie się możliwości wypełniania wektorowego szkieletu (*wireframe*) nieprzeźroczystą płaszczyzną. Dzięki temu to, co w symulowanej przestrzeni znajdowało się "z tyłu" było zasłaniane przez to, co "z przodu". Wynalazek Edwina Catmulla z Uniwersytetu w Utah, czyli *filled vector* oznaczał otwarcie nowego, ekscytującego obszaru eksploracji dla grafików komputerowych. Dalszy rozwój tej dziedziny to powstawanie coraz bardziej złożonych *shaderów* czyli programów opisujących właściwości punktów i wierzchołków w przestrzeni trójwymiarowej, umożliwiających modelowanie oświetlenia, nakładanie tekstur, generowanie cieni, odbić lustrzanych, refrakcji etc. Do najważniejszych *kroków milowych* w tym zakresie należą opracowanie shaderów przez Henriego Gouarda (stąd *gouard shading, g-shading)* w 1971, Bui-Tuong Phonga (Uniwesytet Utah, 1974, *phong shading, p-shading),* a także metod teksturowania przez Eda Catmulla (praca doktorska z 1974, naukowiec ten był założycielem Pixar Animation Studios [1979] i jednym ze współtwórców pierwszego komputerowego filmu animowanego *Toy Story* [1995]).



Ilustracja 21: Przykłady grafiki opartej na 1) *filled vector,* 2) *g-shading,* 3) *p-shading*

Dalsze dzieje grafiki 3D to między innymi opracowanie *bump mappingu,* techniki pozwalającej nadanie "chropowatej" faktury na bazie tekstury nałożonej na obiekt (James

Blinn, 1976) czy też *ray traceingu* polegającego na analizowaniu promieni wirtualnego światła i symulowaniu ich odbić i załamań (Turner Whitted, 1980).

Ilustracja 22: Płaszczyzna z nałożoną teksturą (po lewej) i zastosowanym bump mappingiem (po prawej).

Ilustracja 23: Render bez ray traceingu (po lewej) i po zastosowaniu tej techniki (po prawej).

Wyliczanie istotnych z punktu widzenia rozwoju grafiki komputerowej i demosceny wynalazków, teorii i konceptów można by oczywiście kontynuować bardzo długo. Z perspektywy naszych rozważań istotne jest, że w okresie między latami sześćdziesiątymi a osiemdziesiątymi narodziła się technologia i ufundowane zostały kompletne podwaliny pod szereg zjawisk, które omówione zostaną poniżej. Ostatnim procesem, który musiał mieć miejsce było już jedynie trafienie komputerów do szerokiego odbiorcy, co stało się faktem niedługo później.

Prehistoria

To interesujące, jak w przełomowych momentach narodzin wielu technologii medialnych kluczowym etapem uruchamiającym geometryczne przyśpieszenie ich rozwoju było zainteresowanie ze strony szeroko rozumianej branży rozrywkowej. Miało to miejsce w przypadku narodzin kinematografu, który z technicznej ciekawostki został zamieniony na pełnowartościowe medium przez kilku wizjonerów, którzy potrafili dostrzec jego potencjał u progu dwudziestego wieku - stało się tak również z komputerem, który narodził się jako *poważne* narzędzie badawcze i początkowo nie przewidywano nawet, żeby mógł w dającej się przewidzieć perspektywie trafić do masowego odbiorcy. Już w 1958 roku Higinbotham pracownik centrum badań atomowych postanowił zabawić osoby zwiedzające instytut prezentując proste urządzenie (przełącznik), za pomocą którego można było "odbijać" jasny punkt na niewielkim ekranie analogowego oscyloskopu. Okazało się, że tak trywialna zabawa miała w sobie coś, co powodowało, że trudno się było od niej oderwać (propozycja opatentowania wynalazku spotkała się z krytyką władz Instytutu Badań Jądrowych). Jednak twórcą pierwszej w pełnym tego słowa znaczeniu gry komputerowej był prawdopodobnie Steve "Slug" Russell, pracownik MIT znany również jako twórca pierwszego interpretatora języka Lisp. Zainspirowany *Tenisem dla Dwojga* zaprojektował on program, który umożliwiał sterowanie ruchem niewielkiego obiektu po ekranie komputera. Oryginalny *Spacewar* (1961) został wykorzystany przez firmę DEC do testów i promocji komputera PDP-1. Popularność zdobył ograniczoną ze względu na niewielki nakład urządzenia (wyprodukowano około pięćdziesięciu sztuk, w cenie około 120 tysięcy dolarów każda). Za *pierwszą prawdziwą grę* uważa się *Spacewar* głownie dlatego, że wprowadziła system liczenia punktów, a zatem policzalny wynik i silnik dla rywalizacji - jedną z cech różniących gry od zabawek (na bazie programu powstał późniejszy niewątpliwy hit czyli *Asteroids* firmy Atari [1979]). W 1966 roku na Uniwersytecie w Utah *Spacewar* został dostrzeżony przez studenta Nolana Bushnella, który zrozumiał drzemiący w programiku potencjał. Z jego wykorzystaniem trzeba było jednak poczekać na rozwój i spadek cen podzespołów elektronicznych, które musiały znaleźć się w finansowym zasięgu przeciętnego Amerykanina. Cel udało się osiągnąć się na początku lat siedemdziesiątych wypuszczając na rynek około 1,5 tys. automatów *arcade* z zainstalowaną grą *Computer Space.* Konsola furory nie zrobiła trafiając głównie do barów jako atrakcja stawiana obok flipperów i jednorękich bandytów - okazała się zbyt skomplikowana dla tamtejszego odbiorcy, jednak otworzyła Bushnellowi drogę do otwarcia

najważniejszej inwestycji swojego życia oraz jednej z najistotniejszych dla rozwoju branży - firmy Atari.

the Commie, **pierwsze popularne komputery osobiste,** ***crackscena***

"Nie widzę powodu, dla którego ktoś mógłby chcieć mieć komputer w swoim domu."
Ken Olsen, założyciel DEC (twórców m.in. PDP-1) w 1977

Początki komputerów osobistych to przede wszystkim czasy bardzo dynamicznego rozwoju hardwaru. Technologie elektroniczne ewoluowały w zawrotnym tempie, co doprowadziło do nie mających precedensu zmian cywilizacyjnych. Przełomy miały miejsce tak szybko, że trudno je było śledzić nawet osobom bezpośrednio w nich uczestniczącym. Kiedy Ken Olsen wypowiadał feralne słowa na temat domowych urządzeń komputerowych - mówił z perspektywy dyrektora jednej z największych firm dostarczających potężne (gabarytami) *mózgi elektronowe* dla biznesu i nauki. W tym samym czasie na rynku funkcjonował już kosztujący 400$, wyposażony m.in w napisany przez Billa Gatesa i Paula Allena interpreter Microsoft BASIC kultowy dziś mikrokomputer ALTAIR 8800. Urządzenie to, pozbawione monitora i klawiatury, przeznaczone było do samodzielnego montażu ale wyposażone zostało w procesor Intel 8080 (taktowany 2MHz) oraz pamięć operacyjną 256b... i można je było postawić na biurku, a następnie samodzielnie oprogramować (za pomocą przełączników). ALTAIR uważany jest dziś za jeden z pierwszych komputerów osobistych. Na rok przed niefrasobliwą wypowiedzią Olsena na rynku debiutował też zaprojektowany przez Stevena Wozniaka Apple I (cena 666$, nakład 200 sztuk) z nieco słabszym procesorem ale za to większą pamięcią, klawiaturą (obudowę trzeba było zbudować sobie samemu) i możliwością wyświetlania tekstu w trybie 40x24 znaków. Prawdziwa rewolucja przyszła jednak kilka lat później za sprawą firmy stworzonej przez młodego emigranta z Polski, ocalonego z Holocaustu (był więźniem Auschwitz) - Idka Tramielskiego, znanego w USA jako Jack Tramiel, założyciel firmy Commodore.

Commodore Tramiela i Atari Bushnella wraz z Apple duetu Wozniak/Jobs oraz Microsoftem założonym przez Billa Gatesa i Paula Allena zmieniły całkowicie sposób w jaki najnowsze technologie docierały do szeregowego odbiorcy i doprowadziły do tego, że komputery stały się ogólnodostępnym sprzętem biurowym i osobistym.

Flagowym produktem komputerowym firmy Commodore był model C64 opracowany w 1982 roku na bazie 8 bitowego mikroprocesora MOS Tech 6510 (później 8500) taktowanego na mniej więcej 1MHz. Urządzenie wyposażone było w 64kB RAM i 20kB ROM, układ graficzny VIC-II pozwalał na wyświetlanie obrazu w rozdzielczości 320x200px przy jednobitowej głębi (lub 160x200 przy 2b/piksel itd.). Według specyfikacji system graficzny potrafił wyświetlać do ośmiu sprite'ów (elementów wizualnych) przy 24x21 piksela na sprite. Konstrukcja zawierała jeden z najlepszych układów dźwiękowych w tym czasie: SID pozwalał na syntezowanie muzyki o bardzo charakterystycznym brzmieniu co powodowało, że był chętnie wykorzystywany nawet przez profesjonalnych artystów. Te cechy oraz bardzo dobra cena spowodowały, że C64 to najlepiej sprzedający się komputer w historii (z szacowanym wynikiem między 10 a 17 milionów egzemplarzy, choć niektóre źródła mówią o 22 milionach).

Commodore 64 był również pierwszą platformą, wokół której zaczęła rozgrywać się aktywność scenowa we właściwym tego słowa znaczeniu. Głównym powodem była ogromna popularność tego komputera, a co za tym idzie wielość oprogramowania powstającego do pracy na nim. Na początku lat osiemdziesiątych ubiegłego stulecia nie funkcjonowały jeszcze silne zabezpieczania dotyczące praw autorów softwaru. O ile Europa Zachodnia i USA posiadały pewne regulacje prawne w tym zakresie, o tyle same programy nie były zbyt intensywnie chronione przed samowolnym rozpowszechnianiem. Kopiowanie zakupionych programów w gronie znajomych, dystrybucja za pośrednictwem klubów komputerowych, a nawet sprzedaż pirackich kopii na lokalnych bazarach czy giełdach była czymś całkowicie powszechnym. Ujawniła się przy tym jedna z istotnych, nowych cech medium cyfrowego - kopiowanie między dyskietkami czy taśmami nie powoduje strat jakości. Program przegrany z kasety na kasetę działał dokładnie tak samo dobrze jak oryginał podczas gdy muzyka kopiowana z takiej samej kasety na inną (szczególnie za pomocą amatorskiego sprzętu) brzmiała gorzej. Tym większa była zachęta do dzielenia się kopiami posiadanych gier i programów użytkowych wśród użytkowników pierwszych komputerów osobistych. Problemy w tym zakresie zaczęły się tak naprawdę bardzo wcześnie - znaczącym przykładem (z roku 1975) może być kompilator języka BASIC na urządzenie ALTAIR 8800, którego wydrukowany kod źródłowy został udostępniony członkom Palo Alto Hombrew Computer Club (w którym prezentowali swoje pierwsze dzieła m.in. Wozniak z Jobsem) przez jednego uczestników spotkań. Autor oprogramowania, dziewiętnastoletni wówczas Bill Gates stał się w ten sposób jedną z pierwszych ofiar piractwa komputerowego, jego ówczesny gniew przełożył się na

późniejszą krucjatę przeciwko nielegalnemu dystrybuowaniu programów (co uczyniło go też jednym z ulubionych obiektów drwin i ataków ze strony komputerowego *undergroundu*). Popularność Commodore 64 i postępująca różnorodność powstającego oprogramowania potęgowała niepożądane przez twórców zjawiska. W związku z tym zaczęli oni masowo wprowadzać mechanizmy zabezpieczające zarówno nośniki jak i software. O ile z tymi pierwszymi było stosunkowo łatwo sobie poradzić np. przez zatkanie odpowiedniego otworu w kasecie albo wycięcie żyletką odpowiedniego fragmentu koszulki dyskietki 5,25", o tyle usunięcie tych drugich wymagało pewnej specjalizacji programistycznej. Osoby i grupy wyspecjalizowane w łamaniu oprogramowania nazywano *crackerami.* Funkcjonowali oni w środowisku użytkowników komputerów osobistych obok innych ekspertów, którzy potrafili np. zmieniać kod gry na własne potrzeby czy pisać niskopoziomowe narzędzia dla optymalizacji pracy swoich maszyn - *hakerów*[33]*.*

Crackscena rozwijała się prężnie w latach osiemdziesiątych i dziewięćdziesiątych, łączyła wszak przyjemne - satysfakcję z pogłębionej wiedzy na temat komputerów i technologii bezpieczeństwa, z "pożytecznym" czyli uczestnictwem w społeczności ludzi posiadających dostęp do najnowszego i *złamanego* softwaru. Działalność crackerską szybko ogarnął rodzaj rywalizacji. Grupy zaczęły konkurować w zakresie tego, kto wypuści *release* szybciej, skuteczniej, lepiej - zyskując tym samym większy *fame* w środowisku. Do złamanego przez siebie oprogramowania zaczęto dołączać podpisy informujące o nazwie grupy, czasie wykonania operacji. Tego rodzaju *cractro* (od połączenia słów *intro* oraz *crack*) stawały się coraz bardziej wyrafinowane zawierając stylizowane grafiki z logiem grupy, fantazyjnie zawijające się na ekranie teksty z pozdrowieniami dla zaprzyjaźnionych crackerów (lub wręcz przeciwnie), pogróżki pod adresem korporacji itp. Następnie pojawiły się towarzyszące całości muzyka i animacje.

Ilustracja 24: Logo istniejącej do dziś grupy Fairlight oraz klasyczny tekstowy *scroller w* typowym *cractro*

[33] Pamiętajmy. W języku *sceny* - haker to nie osoba zajmująca się włamywaniem się do systemów komputerowych lecz generalnie mistrz w posługiwaniu się sprzętem i kodem. Jest to określenie pozytywne, oznaczające najwyższy stopień wtajemniczenia w świat informatyki.

Bardzo wielu użytkowników zaczynających swoją przygodę z komputerami w dwóch ostatnich dekadach dwudziestego wieku lepiej kojarzy i rozpoznaje nazwy takich grup i crackerów jak Fairlight, HYBRID, INC (International Network of Crackers), TRSI (Tristar and Red Sector Incorporated), MOTiV8 niż loga firm produkujących właściwe oprogramowanie. Warto zwrócić uwagę, że *cracksena* to nie tylko programiści łamiący oprogramowanie ale również fani, animatorzy spotkań, grupy zajmujące się dystrybucją nośników (taką funkcję pełnił np. MOTiV8 Couriers) w czasach przed masowym dostępem do Internetu.

Ilustracja 25: *Cractro* na platrformę PC autorstwa grupy TRSI

Crack intro miało jeszcze jedną ciekawą cechę, która wynikała bezpośrednio z ograniczeń sprzętowych. Chodzi mianowicie o fakt, że dyskietka z programem miała ograniczoną pojemność (ok. 1200 KB w przypadku nośnika 5,25", ok. 1450 KB dla 3,5") co oznaczało, że cracker chcąc dodać do złamanego programu swój znak musiał zmieścić go obok właściwego softwaru na dysku. Praca twórcy *intra* odbywała się więc w ramach ścisłych ograniczeń, zaś celem było stworzenie nie tylko prezentacji ciekawej ale i maksymalnie zoptymalizowanej. W pewnym momencie okazała się rzecz niezwykła - fani nowych technologii, użytkownicy, piraci i hakerzy zaczęli być zainteresowani *cractrami* samymi w sobie. Społeczność komentowała jakość animacji, użyte efekty - przede wszystkim w kontekście ich technicznego skomplikowania oraz innowacyjności. Uczestnicy giełd pirackiego oprogramowania, spotkań pasjonatów IT czy *copy parties* zaczęli wymieniać się dyskietkami, które zawierały wyłącznie specjalnie wycięte *crack intra.* Stały się one obiektem zainteresowania samym w sobie. Tak narodziły się *dema.*

Demoscena

Jak to zostało już wyżej nakreślone - demo to ogólna nazwa na często bardzo zróżnicowane programy komputerowe działające na wielu platformach, które jednak łączy :

- nieinteraktywność (w tym sensie są to prezentacje);
- fakt bycia wykonywanymi w czasie rzeczywistym przez fizyczne urządzenie (czyli ich prezentowanie np. w postaci nagrania wideo nie jest "zgodne ze sztuką", choć oczywiście w Internecie udostępnia się filmy ze względu na fakt, że wiele dem działa na bardzo egzotycznych, zapomnianych lub modyfikowanych platformach - nie mniej jednak zaprezentowanie w czasie demoscenowego konkursu zrzutu wideo jest absolutnie niedopuszczalne);
- dążenie do krańcowego wyeksploatowania możliwości obliczeniowych platformy;
- dodatkowo: chęć uzyskania interesującego efektu estetycznego (właściwość ta jest pożądana ale nie najważniejsza; często zdarza się tak, że rywalizację wygrywa demo, które wydaje się mniej estetyczne ale za to wykorzystuje jakąś nieznaną dotąd właściwość procesora graficznego czy prezentuje po raz pierwszy unikatowy efekt);

Kiedy forma ta usamodzielniła się z obiektu zależnego od działalności *crackerów* jakim było *intro* grunt pod powstanie nowej nowej subkultury był już bardzo dobrze przygotowany. Ogromne *copy parties* mogły zmienić się w *demo parties* gdyż organizatorzy mieli już doświadczenie w organizowaniu podobnych imprez. Istniejące sieci dystrybucji bardzo szybko rozbudowały się i wyspecjalizowały tworząc nawet specjalną funkcję obok kodera, muzyka czy grafika jaką był *swapper* - osoba kopiująca i rozsyłająca nośniki z produkcjami do innych *swapperów,* którzy przekazywali je dalej (osoby te szczególnie dobrze radziły sobie z hackowaniem... systemu pocztowego - w kręgach *swapperskich* przekazywano sobie wiedzę na temat zabezpieczania znaczków pocztowych tak aby dało się z nich zmyć pieczęć czy preparowania zużytych do ponownego użycia, osobną dziedziną wiedzy był *phreaking* - umiejętność *łamania* zabezpieczeń publicznych telefonów). *Z crackscenowych* fanzinów z łatwością mogły wydzielić się pisma poświęcone wyłącznie (lub głównie) tematom scenowym itd.

Największy swój rozkwit demoscena przeżyła po pojawieniu się na rynku innej niesłychanie ważnej marki ze stajni Commodore - Amigi. C64 i różne rodzaje

mikrokomputerów Atari reprezentowały tanie 8-bitowe komputery osobiste, które po raz pierwszy znalazły się w zasięgu niemal każdego zainteresowanego. Amiga, wyposażona w procesor z rodziny Motorola 68000 (ten sam co w Apple Macintosh) była potężnym skokiem naprzód jeśli chodzi o możliwości obliczeniowe, a przede wszystkim multimedialne domowych stacji roboczych. Najpopularniejszy (przede wszystkim ze względu na cenę) model Amiga 500 oferujący ok. 7MHz procesor, 512 kB RAM i 256 kb ROM (+ gniazda rozszerzeń) pojawił się na rynku w 1987 roku i od razu zrobił wielką karierę. Trwała ona do połowy lat dziewięćdziesiątych kiedy to zaczęła się inwazja architektury PC, zakończona praktycznie całkowitym wyparciem konkurencyjnych rozwiązań na rynku komputerów osobistych (choć np. firma Apple "przesiadła się" z mikroprocesorów typu PowerPC na rozwiązanie Intela [czyli: kompatybilne z PC] dopiero w 2006 roku). Nie mniej jednak Amiga, w różnych swoich wariantach pozostaje do dziś jedną z najbardziej kultowych i lubianych platform dla twórców demoscenowych.

Sztuka komputerowa (?)

Jak to zostało powyżej (w największym skrócie) wykazane, zjawisko demosceny pozostaje w bezpośrednim powiązaniu z historią komputerów osobistych oraz ewolucją systemów graficznych. To powiązanie nie jest przypadkowe, zwraca uwagę na często pomijany fakt, który okaże się bardzo istotny w naszych dalszych rozważaniach - kiedyś użytkownicy komputerów byli też programistami w ścisłym tego słowa znaczeniu. Korzystanie z takich, a nie innych narzędzi zdeterminowało ich sposób myślenia o świecie nowych mediów - a nade wszystko wpłynęło na preferowane przez nich formy ekspresji.

Po okresie bardzo dynamicznego rozwoju w latach dziewięćdziesiątych, naznaczonych rywalizacją Amigi z PC demoscena ustabilizowała się w coś, co można by nazwać spójnym ruchem o charakterze z pogranicza subkultury (wskazuje na to specyficzny język, silnie tożsamościowy charakter relacji między twórcami a środowiskiem, swoisty etos) i sztuki (wybór preferowanego medium ekspresji, silny nacisk na specyficzną estetykę, bezinteresowny charakter działalności [warto zwrócić uwagę: dema komercyjne w zasadzie nie istnieją]). Wśród dzisiejszych *scenowców* kwestia włączania ich działalności w zakres kultury i sztuki elektronicznej nie budzi tak powszechnego sprzeciwu - prawdopodobnie ze względu na dowartościowanie tego rodzaju ekspresji jakie miało miejsce w ciągu ostatnich kilkunastu lat. Cały czas fundamentalnie ważne pozostaje traktowanie demoscenowych *produkcji* jako przede wszystkim tworów o charakterze

informatycznym, programistycznym i algorytmicznym. Na największych, dorocznych *demo parties* (takich jak Breakpoint/Revision [Easterparty], Assembly, Evoke, the Gathering, Mekka&Symphosium)

Ilustracja 26: zdjęcie z niemieckiego *party* Breakpoint (fot. J.K.A.).

są one prezentowane najczęściej w kilku podstawowych kategoriach, które podkreślają ich ścisły związek z platformą (np. C64, ZX Spectrum, Atari ST, Amiga, PC), charakterem kodu (np. intro 4K, intro 8K, intro 64K, demo). Osobno funkcjonują też produkcje poza klasycznymi zakresami takie jak crazy/wild demo (zrobione dla rozrywki, humorystyczne dema na nietypowe i egzotyczne platformy), birthtro (intro oznajmujące narodziny dziecka scenowca), invite (demo-zaproszenie na *party*) itp.

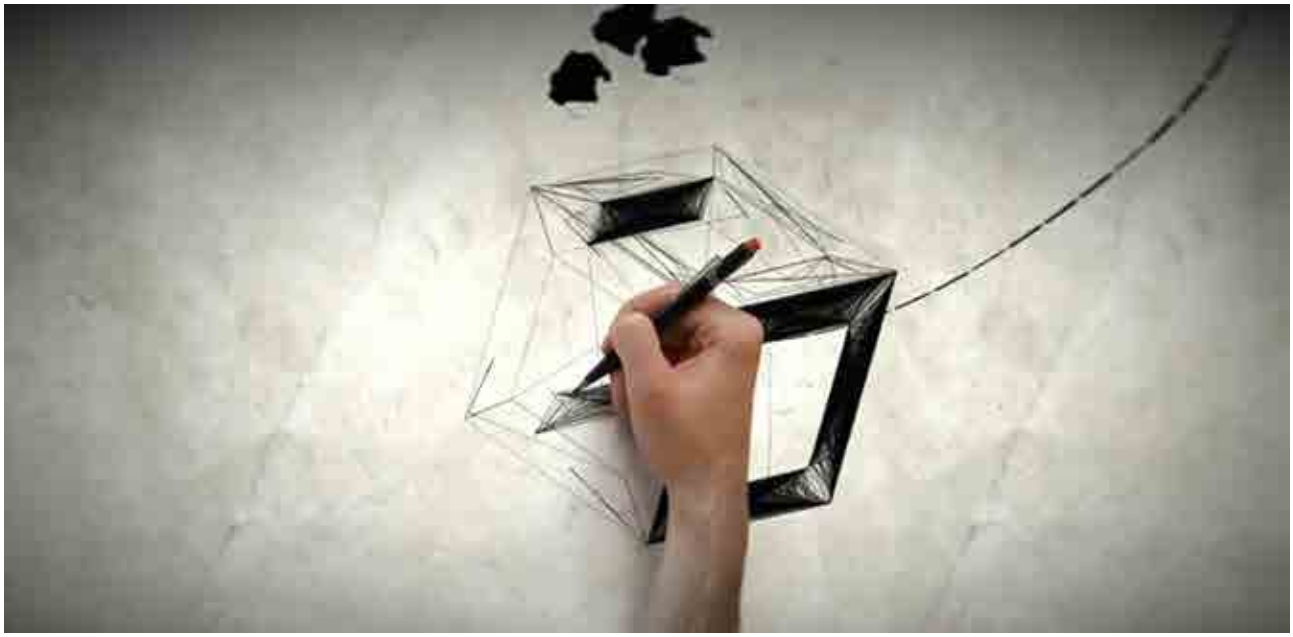
Z perspektywy studiów nad oprogramowaniem demoscena całościowo jawi się jako zjawisko niezwykłe i interesujące. Bowiem - jak to się za chwilę okaże - z perspektywy tej dziedziny wiedzy oprogramowanie interesujące jest przede wszystkim jako system regulujący przetwarzanie, przepływ i dostęp do informacji (w tym, a może - głównie, danych kulturowych). Logika oprogramowania jest więc podskórną logiką działania zapośredniczonej cyfrowo kultury i sztuki. Demoscena łamie ten sposób patrzenia na technologię, dla jej twórców pojęcie medium wraca do swojego znaczenia sprzed ery

komunikacyjnego Wielkiego Wybuchu. Komputer jest dla kodera, grafika, muzyka narzędziem i tworzywem. Chodzi zaś, owszem, o stworzenie czegoś interesującego estetycznie ale w ramach narzuconych platformowo ograniczeń. Dema powstające na różnych urządzeniach wykorzystują ich możliwości (np. specyficzne brzmienie SIDa) ale też przekraczają je (np. w demach C64 za pomocą złożonych technik udaje się uzyskać powyżej stu dwudziestu sprite'ów zamiast założonych przez twórców urządzenia ośmiu).

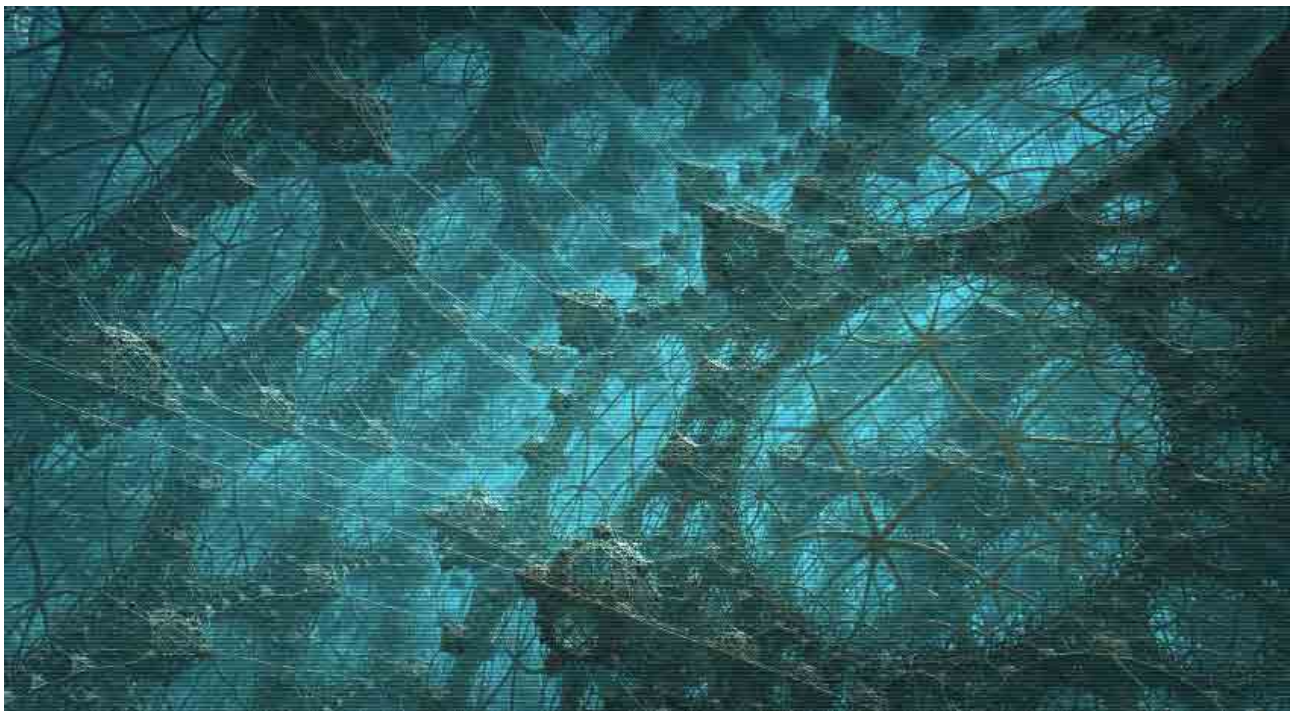
Ilustracja 27: demo *Snapshot* grupy *Glance* na platformę C64, zawiera elementy grafiki 3D (!) [2010].

Ilustracja 28: demo polskiej grupy *Elude* pod tytułem *We come in peace* napisane na Amigę w 2010, zawiera grafikę 3D, *shadeing*, transparencję i cały szereg niespotykanych dotąd na tej platformie efektów.

Jeśli więc zgodzimy się nazwać omawiany rodzaj twórczości sztuką, możemy również bez wahania stwierdzić, że jest to sztuka najbliższa medium komputerowemu jaką możemy sobie wyobrazić. Zarówno na poziomie przywiązania do hardwaru jak i oddania sprawiedliwości softwarowemu aspektowi pracy urządzeń obliczeniowych.



Ilustracja 29: *Ms. Eshers Nightmare* grupy *Andromeda Software Developement* (PC demo) [2014]



Ilustracja 30: *Infinite Landscape* - grafika proceduralna autorstwa *Demoscene Passivist* [2010]

Αναλυτικά 3:
Od Assemblera przez Lisp i Prolog po Wolfram Language i Swift. Poziomy języków programowania.

"Programowanie w kodzie maszynowym przypomina jedzenie wykałaczką. Bity są tak małe, a cały proces tak mozolny, iż wydaje się, że obiad trwa w nieskończoność. Bajty kodu maszynowego realizują najmniejsze i najprostsze spośród możliwych do wyobrażenia zadań obliczeniowych - ładowanie liczby do pamięci procesora, dodawanie jej do innej i zapisywanie w pamięci - trudno nawet sobie wyobrazić w jaki sposób może się z tego składać jakieś dane" (Petzold, 2002, p. 399)

"Wyrażenia »to zawiera się w całości w czymś innym« i »to drugie orzeka o wszystkich przypadkach pierwszego« stwierdzają to samo. Mówimy, że »coś się orzeka o wszystkich przypadkach«, gdy nie można wskazać żadnego przypadku, objętego przedmiotem, o którym nie orzekałby ten drugi termin; wyrażenie: »nie być orzekanym o żadnym« należy rozumieć w ten sam sposób." (Arystoteles, 1990, p. 129)

"Kod źródłowy:

```
if count = 10
     goto end-of-job
else
     goto compute-again
endif
```

Język assemblera / Język maszynowy:

```
compare A to B      /    compare 3477 2883
if euqal go to C    /    if = go to 23732
go to D             /    go to 23119
```

Kod maszynowy:

```
10010101001010001010100
10101010010101001001010
10100101010001010010010"
```

(PC Magazine Encyclopedia, definition of: *compiler*)[34]

[34] http://www.pcmag.com/encyclopedia/term/40105/compiler (dostęp: 1.08.2014)

Wiemy już (mniej więcej) czym jest komputerowe oprogramowanie, jakie są jego korzenie, a także znamy podstawowe konstrukcje, z których jest zbudowane. Udało się też przyjrzeć kilku jego artystycznym zastosowaniom. Zanim przejdziemy do kwestii związanych z analizą całkiem współczesnych konstrukcji informatycznych i abstrakcji na nich zbudowanych - przyjrzyjmy się jeszcze ewolucji tego medium na wybranych przykładach. Spowoduje to, że łatwiej nam będzie zrozumieć aktualne dyskursy związane z jednej strony z programistycznymi ontologiami, z drugiej strony z napędzanymi przez ludzi (*human driven)* dynamikami pracy softwaru.

A 3.1 Prawdziwi Programiści

W redagowanym przez Erica Raymonda *Jargon File* - stronie internetowej będącej swoistym słownikiem pojęć i etosu hakerskiego[35] (mającym też swoją drukowaną wersję znaną jako *The New Hacker's Dictionary)* czytamy, że Prawdziwy Programista to:

"Specyficzny podgatunek hakera, zachowujący nonszalanckie podejście do złożoności, które graniczy z arogancją, nawet jeśli jest usprawiedliwione doświadczeniem. Archetypowy Prawdziwy Programista lubi programować w *czystym metalu* (czyli na przełącznikach, samym hardwarze bez użycia nawet asemblerów - dop. J.K.A.) i jest w tym dobry. Pamięta binarne kody operacji (*opcode)* dla każdej maszyny, na której kiedykolwiek programował, uważa że języki wysokiego poziomu (*HLL - High Level Language)* są dla *cieniasów* i używa debuggera do pisania kodu ponieważ pełnoekranowe edytory są dla *słabych*. Prawdziwi Programiści nie są zadowoleni z kodu, który nie jest tak skondensowany, że znajduje się na skraju załamania. Prawdziwi Programiści nie używają komentarzy ani nie piszą dokumentacji: »jeśli to było trudne do napisania« - mawiają - »powinno być też trudne do zrozumienia«. Prawdziwi Programiści potrafią zmusić maszyny do robienia rzeczy, których nigdy nie miały w specyfikacji; tak naprawdę, nie są oni nigdy zadowoleni dopóki im się to nie uda. Kod Prawdziwego Programisty może zatrwożyć swoją złowrogą znakomitością nawet jeśli wydaje się *czerstwy*. Prawdziwi Programiści żyją na śmieciowym jedzeniu i kawie, wieszają grafiki z drukarek igłowych na ścianach i przerażają innych programistów - ponieważ kiedyś, ktoś być może będzie

[35] http://www.catb.org/jargon/ (dostęp 1.08.2014)

musiał zrozumieć ich kod aby coś w nim zmienić. Ich następcy generalnie uważają, że do dobrze, że nie ma już zbyt wielu Prawdziwych Programistów." [36]

Prawdziwi Programiści, tacy jak opisywani w cytowanym tekście w zasadzie zniknęli z krajobrazu pod koniec lat osiemdziesiątych dwudziestego wieku - urządzenia komputerowe stały się zbyt złożone, zamknięte, a zadania przez nie wykonywane zbyt skomplikowane aby programować je w *czystym metalu*. Komputery przestały być tym, co nazywano pieszczotliwie *żelastwem* i oklejano parodiującymi nazistowską propagandę plakatami "*Blinkenlights*". Jednak sposób myślenia pokolenia PP, podejście do rozwiązywania problemów oparte na przeświadczeniu, że jeśli jakieś rozwiązanie jest uznawane za niemożliwe to prawdopodobnie po prostu nikt się nim jeszcze poważnie nie zainteresował, przetrwało w postaci różnego rodzaju współczesnych hakerów. Prawdziwi Programiści byli ogniwem łączącym świat *developerów* ze światem inżynierów i fizyków, którzy jako pierwsi zajmowali się technologiami komputerowymi. Należeli do nich Ekcert i Mauchly - twórcy kultowego ENIAC'a lecz także tacy naukowcy jak Richard Feynman, który biorąc w młodości udział w pracach nad amerykańską bombą atomową w ramach Projektu Manhattan w laboratoriach Los Alamos zajmował się m.in. opracowaniem systemu wykorzystującego z karty dziurkowane IBM na potrzeby obliczeń fizycznych, testując - swoim zwyczajem - niezwykłe i wydawałoby się szalone nieraz rozwiązania.

W naszych rozważaniach przyszedł moment kiedy musimy przejść od refleksji dotyczących logiki, matematyki i podstawowych układów elektronicznych - tego czym tak naprawdę zajmowali się Pierwsi Programiści do tak nielubianego przez nich programowania wysokopoziomowego, gdyż to ono stanowi większość tego co współcześnie dzieje się w świecie kodu.

Ilustracja 31: plakat "Blinkenlights" (jeden z licznych wariantów),
żart w hackerskim stylu z czasów Pierwszych Programistów

[36] http://www.catb.org/jargon/html/R/Real-Programmer.html (dostęp: 1.08.2014)

A 3.2 BASIC(s) czyli podstawy.

BASIC czyli Beginners All-purpose Symbolic Instruction Code to język (a w zasadzie rodzina języków) programowania wysokiego poziomu (za chwilę wyjaśnimy to pojęcie) opracowana w 1964 roku przez Thomasa Kurtza i Johna Kemeny'ego na bazie Fortrana II i elementów Algola 60.

Pierwszy z wymienionych języków, stworzony w połowie lat pięćdziesiątych XX wieku był pionierski jeśli chodzi o wspomnianą wyżej właściwość bycia wysokopoziomowym. Oznaczało to, że jego składnia i wyrażenia starały się być możliwie najbliżej języka naturalnego a w efekcie być jak najłatwiej zrozumiałymi dla człowieka. W przeciwieństwie do asemblerów, w językach HLL pisze się (prawie) zawsze w ten sam kod dla różnego rodzaju urządzeń z tej samej rodziny co oznacza, że tworzone programy są bardziej uniwersalne, zaś programista jest w mniejszym stopniu obciążany koniecznością zważania na konkretny hardware na jakim ma działać program. Skutkiem ubocznym takiego stanu rzeczy jest konieczność przeprowadzania *kompilacji* programu przed jego uruchomieniem. Kompilacja to proces automatycznej translacji kodu źródłowego (w języku wysokiego poziomu) na kod wyjściowy w języku maszynowym (czasami z etapem pośrednim asemblera). Poziomy abstrakcji wprowadzane w różnych językach programowania mogą być oczywiście różne.

Drugi język, na bazie którego powstał BASIC - Algol (ALGOrithmic Language) to twór nieco późniejszy. Odegrał jednak ogromną rolę w historii programowania głównie dlatego, że jego wersja Algol-60 wprowadzała nieznane dotąd konstrukcje takie jak instrukcje blokowe, rekurencję, tablice dynamiczne czy definiowane typy danych.

Ufundowany na takich podwalinach BASIC stał się jednym z najpopularniejszych narzędzi pracy naukowców i inżynierów w latach sześćdziesiątych i siedemdziesiątych. Musimy pamiętać, że w tamtych czasach niemal każde użycie komputera wiązało się z koniecznością opracowania własnego softwaru - ilość ogólnie dostępnych, uniwersalnych programów przydatnych w pracy badawczej była potężnie ograniczona. Jednocześnie nie każdy z potencjalnie zainteresowanych wykorzystaniem komputera użytkowników miał wymagane do poważnego programowania umiejętności matematyczne oraz wiedzę z zakresu budowy urządzeń elektronicznych. BASIC miał się stać lekarstwem na te bolączki.

Kolejne jego wersje i kompilatory powstawały więc na coraz szerszy zakres urządzeń, aż wraz z początkiem ery komputerów osobistych trafiły pod strzechy osób zainteresowanych informatyką. Jak pamiętamy z fragmentu poświęconego demoscenie jednym z pierwszych komputerów osobistych był ALTAIR-8800 zaś programem, na którym został popełniony "grzech pierworodny" piractwa był właśnie kompilator BASIC'a na tą platformę stworzony przez Billa Gatesa i Paula Alena. David Bunnell - jeden z pionierów komputeryzacji jest autorem stwierdzenia, że kluczami do rewolucji cyfrowej była dostępność urządzeń obliczeniowych oraz ich zrozumiałość. ALTAIR wypełnił pierwsze, zaś BASIC drugie z tych kryteriów. Język programowania towarzyszył *boomowi* komputerowemu lat osiemdziesiątych i dziewięćdziesiątych będąc obecnym w różnych wariantach na wszystkich ważnych platformach tego okresu (w tym oczywiście C64, Amidze), a także wkroczył wraz z systemem operacyjnym Microsoftu w erę PC gdzie ewoluował do postaci Visual Basica. Dziś nie należy on już do popularnych sposobów tworzenia oprogramowania - został zastąpiony konstrukcjami o dużo wyższym stopniu abstrakcji i mniejszej liniowości. Nie mniej jednak BASIC pozostaje dla pierwszego pokolenia użytkowników komputerów osobistych czymś podstawowym, kodem, który przepisywali z wypiekami na twarzy z czasopism poświęconych informatyce aby uruchomić prostą grę czy niewielki program użytkowy. W świadomości pokoleń "komputerowców" lat 70-80 doświadczenie BASIC'a pozostaje czymś niesłychanie istotnym.

Z naszej perspektywy najistotniejsze w tym momencie jest wejście na arenę prowadzonych rozważań struktur, które przestają być językiem tylko w sensie metaforycznym a coraz bardziej zaczynają przypominać zjawisko, które nazywamy w ten sposób intuicyjnie. O ile w asemblerach (a już całkowicie w językach maszynowych) trudno jest dopatrzeć się śladów ludzkiej mowy i pisma (to, że są one tam obecne wiemy raczej ze szczegółowych rozważań prowadzonych w poprzednich rozdziałach niż z prostych spostrzeżeń) o tyle BASIC i inne HLL znacznie bardziej przypominają swoistego rodzaju *wyrażenia*.

Dla przykładu. Program wyświetlający tekst "Hello world" w asemblerze 6502 (podobny do używanego m.in. Commodore C64, Apple II) wygląda tak:

```
          .or $300
main      ldy #$00
.1        lda str,y
```

```
          beq .2
          jsr $fded
          iny
          bne .1
.2        rts
str       .as "HELLO WORLD"
          .hs 0D00
```

zaś w BASIC'u następująco:

```
          PRINT "HELLO WORLD"
```

Jak widzimy, różnica w przystępności (i zwięzłości) kodu jest kolosalna. Programista zamiast dopasowywać swój sposób myślenia do konstrukcji konkretnego urządzenia (w kodzie asemblera musimy znać konkretne adresy sprzętowe) po prostu wydaje mu polecenia.

Więcej przykładów kodu źródłowego w BASICu oraz szersza refleksja nad jego specyfiką znajduje się w rozdziale A5 na temat *Software Criticism.*

A 3.3 Hakerzy i malarze.

Ewolucja języków programowania, którą powyżej w zarysie prześledziliśmy spowodowała znaczącą zmianę w relacji użytkownika i komputera. Z inżynierskiego narzędzia służącego automatyzacji pewnych skomplikowanych zadań - w szczególności obliczeń stały się one środkami swoistego rodzaju ekspresji. Zmianę tą zauważa Paul Graham - eseista, programista i przedsiębiorca, popularyzator Lispu (języka programowania) i twórca jednej z pierwszych aplikacji webowych (działających na serwerze dostępnych przez interfejs przeglądarki internetowej). W swojej książce dzieli się spostrzeżeniami na ten temat, odwołując się do znanej nam już figury hakera jako uzdolnionego programisty, którego tu nazywa wprost - twórcą, a nawet artystą:

"Gdy otrzymałem dyplom ukończenia wydziału nauk informatycznych, postanowiłem zacząć studiować malarstwo. Wiele osób było zaskoczonych faktem, że

ktoś interesujący się komputerami interesuje się również malarstwem. Okazało się, że ludzie myślą, że programowanie i malarstwo to dwie dyscypliny zupełnie od siebie odmienne - że programowanie jest zimne, precyzyjne i metodyczne, a malarstwo to rozbuchana ekspresja jakiejś pierwotnej potrzeby.

Mylili się w obu przypadkach. Programowanie i malowanie mają ze sobą wiele wspólnego. Faktycznie spośród rozmaitych typów ludzi, jakie było mi dane poznać hakerzy i malarze byli do siebie najbardziej podobni.

Tym co łączy hakerów i malarzy, jest fakt, że zarówno jedni jak i drudzy są twórcami. Podobnie jak w przypadku kompozytorów, architektów i pisarzy celem przyświecającym hakerom i malarzom jest tworzenie wspaniałych rzeczy"

(Graham, 2004, p. 29)

Wobec tak wyrażonej postawy nie dziwi fakt, że ulubionym i promowanym przez tego programistę językiem jest Lisp. Historia Lispu, który właściwiej jest nazwać raczej rodziną języków programowania niż pojedynczym tworem sięga czasów - z punktu widzenia informatyki - zamierzchłych czyli końca lat pięćdziesiątych ubiegłego stulecia. Pierwsza poważna publikacja opisująca potencjalną konstrukcję tego języka to artykuł Johna McCarthy'ego z 1960 roku pt. *Rekursywne funkcje wyrażeń symbolicznych i ich maszynowe obliczanie, część pierwsza* z czasopisma *Communications of the ACM,* w którym podjął on nie tyle misję stworzenia bezpośredniego narzędzia dla programistów lecz pokazania jak zbudować język, który będzie funkcjonalną alternatywną dla niewygodnej w implementacji Maszyny Turinga:

"»Kolejnym sposobem okazania, że Lisp był lepszy niż maszyny Turinga, było napisanie uniwersalnej funkcji Lispa[37] i udowodnienie, że jest krótsza i bardziej zrozumiała niż opis uniwersalnej maszyny Turinga. To była funkcja *eval...* wyliczająca wartość wyrażenia...«. Napisanie *eval* wymagało wymyślenia notacji reprezentującej funkcje Lispa jako dane Lispa i taka notacja została stworzona na potrzeby zapisywania na papierze, bez założenia, że będzie stosowana do zapisywania prawdziwych programów w Lispie."

(Graham, 2004, p. 220)

---

[37] W cytowanym tłumaczeniu książki Grahama dopełniacz od "Lisp" brzmi "Lispa", natomiast szerzej przyjętą w literaturze formą jest "Lispu" - takiej też używam w swoim tekście, pozostawiając mniej poprawną jedynie w cytatach.

Pierwszej implementacji koncepcji McCarthy'ego dokonał w jeszcze w tym samym roku gdy została stworzona, na dwa lata przez formalną publikacją opisującego ją artykułu Steven "Slug" Russell (ten sam, którego wspominaliśmy w poprzednim rozdziale jako autora *Space Wara*). Pełny kompilator Lispu powstał na MIT w 1962 roku.

Co jest takiego niezwykłego w opracowanym pół wieku temu języku, że mimo potężnego skoku technologicznego jaki dokonał się od czasu jego powstania, kolejne wersje Lispu cieszą się tak dużym zainteresowaniem? Po pierwsze, rzecz zasadza się w jego zakorzenieniu w aparacie matematycznym, który jest odporny na upływ czasu. Po drugie pojawia się w nim szereg innowacyjnych koncepcji, które znów wylicza Graham:

- istnienie konstrukcji warunkowych typu *if-then-lese* zamiast dotychczasowego *goto*;
- w Lispie funkcje są typami danych (!);
- język ten jako pierwszy wprowadził funkcje rekurencyjne (!);
- posiada on dynamiczna kontrolę typów co oznacza, że typ jest zależny od zawartości jaką przechowuje zmienna (i może być różny w różnych miejscach programu);
- *garbage collection* czyli usuwanie z pamięci nie używanego już kodu;
- Lisp nie odróżnia wyrażeń od instrukcji (*statement* / *expression*); program to drzewo wyrażeń, z których każde zwraca wartość;
- istnieje w nim osobny typ symboli (jako wskaźników do *hash table*);
- Lisp nie wymaga de facto kompilacji, tj. może działać w czasie edycji - ta cecha czyni go również interesującym narzędziem ekspresji artystycznej (o czym poniżej);

Wyżej wymienione właściwości czynią język stworzony przez McCarthy'ego i Russela niezwykle istotnym z perspektywy historycznej. Tym zaś, co powoduje, że jest on interesujący po dziś dzień jest jego spowodowana konstrukcją elastyczność:

"Lisp nie wygląda dziwnie dlatego, że ma dziwną składnię, ale raczej dlatego, że nie ma jej wcale. Wyrażasz programy bezpośrednio w drzewach analizy składniowej, które są w budowane w tle. Gdy kod napisany w innym języku jest parsowany, drzewa te są zbudowane z list, które z kolei w Lispie są strukturami danych" (Graham, 2004, p. 224)

"Dziwność" składni Lispu polega na tym, że składa się ona w zasadzie praktycznie wyłącznie z nawiasów gdyż - jak to było wyżej wspomniane - wyrażenia (*s-wyrażenia)* są zapisywane jako listy. Pierwszy ich element to nazwa, pozostałe to argumenty:

```
(+ 1 2 3) // zwróci 6
(* 1 (+ 2 (- 4 5))) // jest równoważne algebraicznemu: 1 * (2 + (4-5))
(setq x 10) // nadaje zmiennej x wartość 10
```

inne przykłady kodu w Lispie będą wyglądały podobnie; wyrażenia możemy zagnieżdżać, stosować instrukcje warunkowe etc.:

```
(if (warunek) jeśli-prawda jeśli-fałsz)
```

Dodatkowo ważna jest obecność w omawianym języku tzw. *makr,* będących transformacją jednego kodu w inny za pomocą kodu (jakkolwiek dziwnie by to nie brzmiało). Nie wdając się w szczegóły konkretnych rozwiązań - konstrukcje te pozwalają na "wbudowywanie" syntaktyki innych języków w Lispa. Amorfizm ten prowadzi do sformułowania znanego jako Dziesiąta Zasada Programowania Greenspuna, które stwierdza, że każdy dostatecznie złożony program napisany w Fortranie i C (a więc językach wywodzących się z alternatywnego pnia niż Lisp, *imperatywnych*) zawiera skonstruowaną ad hoc, pełną błędów implementację połowy Common Lispa. Reguła sformułowana w sposób żartobliwy (choćby dlatego, że nie było żadnych dziewięciu wcześniejszych Zasad Programowania) jest świadectwem specyficznego, hackerskiego poczucia humoru - jest prawdziwa[38].

Lisp jest więc przypadkiem szczególnym - językiem opracowanym na bazie zasadniczo nie przeznaczonej do implementacji koncepcji, charakteryzującym się ogromną w stosunku do innych elastycznością. Powoduje to oczywiście liczne problemy przy próbach jego użycia w ramach masowej produkcji typu korporacyjnego. Programując w Lispie każdorazowo definiujemy sobie "świat", w ramach którego chcemy działać i budujemy sobie de facto własny język jego opisu. Z różnych względów może to być kłopotliwe dla firm zajmujących się produkcją komercyjnego oprogramowania - aczkolwiek zważywszy na ogromną ilość dialektów Lispu - nie jest to przeszkoda nie do pokonania. To, co jest problematyczne dla jednych stanowi atrakcję dla innych. Dlatego Lisp bywa postrzegany jako język hakerów (których nadal prezentujemy tu jako tych, którzy szukają komputerowego "oświecenia", chcą zrozumieć naturę, możliwości i ograniczenia tego

[38] vide: http://www.catb.org/jargon/html/H/ha-ha-only-serious.html (dostęp 10.08.2014)

medium - nie "komputerowych włamywaczy"). Eric Raymond w artykule *How to become a Hacker* rekomendując naukę różnych języków programowania napisał:

"LISP jest godny polecenia z innego powodu - poczucia głęboko oświecającego zrozumienia, którego doświadczysz kiedy już go pojmiesz. To poczucie uczyni cię lepszym programistą do końca twoich dni, nawet jeśli nie będziesz używać samego LISPu na co dzień."[39]

Istnieje jeszcze jeden - szczególnie dla nas interesujący powód, żeby docenić ten język programowania. Chodzi o sygnalizowaną od początku rozdziału siłę programowania jako medium. Potężna moc ekspresji Lispu w wymiarze informatycznym (co oznacza zakres idei jakie mogą być w nim wyrażone) idzie w parze z wymiarem, który powinniśmy nazwać *artystycznym* - możliwościami realizowania wartości estetycznych.

Przykład: Lisp - sztuka i oprogramowania. *Live coding.*

Jeśli do programowania w Lispie użyjemy odpowiedniego, realizującego kod w czasie rzeczywistym środowiska, to efekty naszej pracy zostaną wykonane dokładnie w momencie kiedy uniesiemy palce znad klawiatury. Takim interfejsem jest stworzone dla systemu operacyjnego MacOS przez Andrew Sorensena *Impromptu.* Zawiera ono m.in. biblioteki pozwalające na realizowanie wizualizacji 2D i 3D oraz audio przez pisanie kodu w czasie rzeczywistym. Np. po zdefiniowaniu wyrażenia *piano* możemy "zagrać" nutę za pomocą kodu[40]:

```
(play-note (now) piano 60 80 *second*)
```

możemy też zdefiniować akord:

```
(define chord
   (lambda ()
      (play-note (now) piano 60 80 *second*)
      (play-note (now) piano 64 80 *second*)
      (play-note (now) piano 67 80 *second*)))
```

[39] http://www.catb.org/esr/faqs/hacker-howto.html (dostęp 10.08.2014)

[40] przykłady kodu za: http://impromptu.moso.com.au/ (dostęp 10.08.2014)

i wiele innych, bardziej złożonych struktur muzycznych.

Posiadający wiedzę i słuch muzyczny oraz umiejętności programistyczne artysta jest w stanie grać/implementować muzykę o bardzo wysokim stopniu skomplikowania, na wiele instrumentów jednocześnie. Twórca *Impromptu* Andrew Sorensen sam komponuje i wykonuje utwory pianistyczne realizowanie w Lispie[41].

Od początku swojego istnienia Lisp był też bardzo chętnie wybieranym językiem przez programistów zajmujących się sztuczną inteligencją. Było tak ze względu na jego, omawianą wyżej, elastyczność - co czyni go wyśmienitym narzędziem do prototypowania. Ponadto nie bez znaczenia jest jego sposób wspierania programowania symbolicznego, co ma duże znaczenie dla A.I. (Lisp był pot tym względem bardzo długo językiem unikatowym). Nie bez wpływu pozostaje tu też historyczna kwestia zasług jakie położył na polu badań nad sztuczną inteligencją "ojciec" Lispu John McCarthy. Jest on uważany za osobę, która stworzyła pojęcie "A.I." w połowie lat pięćdziesiątych XX wieku.

Jednym z istotnych problemów początków technologii obliczeniowych oraz przedmiotem wielkiej fascynacji ze strony filozoficznie zorientowanych informatyków była kwestia tego, na ile Maszyna Turinga modeluje sposób funkcjonowania ludzkiego umysłu lub - jeśli sformułować problem *słabiej* - czy za pomocą komputera da się modelować pewne aspekty ludzkiej inteligencji. W drugim przypadku odpowiedź była oczywiście twierdząca - sposób działania urządzeń obliczeniowych pod pewnymi względami przypomina nasze myślenie. Obok "syntetycznych" języków programowania tworzonych z oczywistą intencją jak najlepszego współdziałania z takim a nie innym dostępnym hardwarem istnieje też cały szereg tego rodzaju konstrukcji, które mają za zadanie modelować różnego typu działania - w tym przetwarzanie informacji na sposób językowy, logiczny etc. Wiemy, że informatyka wiele zawdzięcza szczególnie tym dwóm ostatnim obszarom refleksji: językoznawstwu (od Algoritmiego i hermeneutów islamskich, przez filozofów-naturalistów poszukujących *lingua universalis*, po Chomsky'ego, którego koncepcję gramatyk generatywnych omawialiśmy) oraz logice matematycznej (w

[41] przykładowe wykonanie utworu: http://en.wikipedia.org/wiki/Impromptu_(programming_environment)#Sample_performance (dostęp 10.08.2014)

szczególności w okresie od russellowskiej rewolucji do dziś). Dlatego warto zwrócić w tym miejscu na inny specyficzny, obok Lispu, sposób realizowania zadań przez języki programowania jakim jest *programowanie w logice (logic programming)* - jedną z najciekawszych realizacji tego konceptu jest język Prolog. Grzegorz J. Nalepa tak o nim pisze:

"Jest do deklaratywny język programowana wysokiego poziomu stworzony na potrzeby budowania systemów inteligentnych. (...) Pierwsza prototypowa implementacja została stworzona przez Alana Colmeraura na uniwersytecie w Marsylii w latach 70. XX wieku. (...) Systemy prologowe są sukcesywnie rozwijane i rozszerzane. Obecnie dostępny jest szereg implementacji, w tym otwarte SWI-Prolog i YAP Prolog." (Nalepa, 2011, p. 385)

Kod Prologu składa się z bazy faktów oraz reguł. Można w nim konstruować predykaty typu: `matka(Anna, Piotr)` służące nadawaniu znaczenia wypowiedziom o zdefiniowanym świecie, tu: "Anna jest matką Piotra". Specyficzne, prologowe predykaty mogą też realizować inne zadania niż opis np. `write('kot')` technicznie też jest "predykatem" tego języka ale skutkuje wypisaniem słowa "kot" na ekranie. Najważniejsze jednak jest to, że system potrafi przeprowadzać *rozumowania* w oparciu o rachunek predykatów pierwszego rzędu (a właściwie o klauzule Horna). Dysponując bazą faktów:

```
ogon(Blekot).
cztery_lapy(Blekot).

ogon(Cwirek).
skrzydla(Cwirek).
dziob(Cwirek).

kot(X) :-
     ogon (X),
     cztery_lapy (x).

poluje (X,Y) : -
     kot(X),
     wrobel(Y).
```

```
wrobel(X) :-
     skrzydla(X),
     ogon(X).

dzwiek(kot, miauczy).
dzwiek(wrobel, cwierka).

zwierze (X) :-
     kot(X).
zwierze(X) : -
     wrobel(X).
```

możemy zadawać pytania typu:

```
?- ogon(Blekot).     // czy Blekot ma ogon?
true.                // program zwraca: "tak"
?- kot(Blekot).      // czy Blekot jest kotem/
true.                // program zwraca: "tak"
```

a także przeprowadzać wnioskowania w rodzaju: "jeśli X ma ogon to jest zwierzęciem" albo "jeśli X ma ogon i miauczy to jest kotem, a skoro jest kotem to jest zwierzęciem i poluje na wróble" (zmodyfikowane przykłady z: Nalepa, 2011, p. 386).

Jak łatwo zauważyć cały system opiera się na założeniu, że aby generować znaczące odpowiedzi urządzenie nie musi w sensie ścisłym *rozumieć* treści symboli, którymi manipuluje. Myślenie może być dzięki temu ujmowane w kategoriach procesu komputacyjnego co, zauważmy, nie oznacza wcale, że ludzki mózg - czy jeszcze bardziej abstrakcyjnie - umysł, działa w sposób analogiczny do komputerowego procesora. Chodzi tylko o to (i aż o to), że proces myślenia (w szczególności - rozumowania) może być rozumiany jako mechaniczne *przetwarzanie symboli.* Pomysł bardzo bliski omawianym uprzednio koncepcjom Leibniza et al.

Ta krótka prezentacja Prologu, włączona do naszych rozważań przy okazji pojawienia się nauki o sztucznej inteligencji pokazuje, że języki programowania mogą się

bardzo różnić nie tylko na poziomie samej składni lecz również w zakresie koncepcji, które napędzają je do działania czy też celów, którym mają służyć. Przy czym niezwykłość sytuacji polega na tym, że cała ta różnorodność realizowana jest cały czas przez w gruncie rzeczy to samo urządzenie (taką czy inną implementację Maszyny Turinga). Różnice między narzędziami programistycznymi to jednak nie tylko pragmatyczna kosmetyka mająca na celu ułatwić twórcom projektowanie rozwiązań mających na celu efektywne rozwiązanie pewnej klasy problemów. Różnorodność języków programowania to przede wszystkim skutek zróżnicowanych poziomów abstrakcji oraz paradygmatów, w ramach których funkcjonują. Zanim zajmiemy się tymi drugimi przyjrzyjmy się jeszcze dwóm interesującym przypadkom języków (bardzo) wysoko poziomowych.

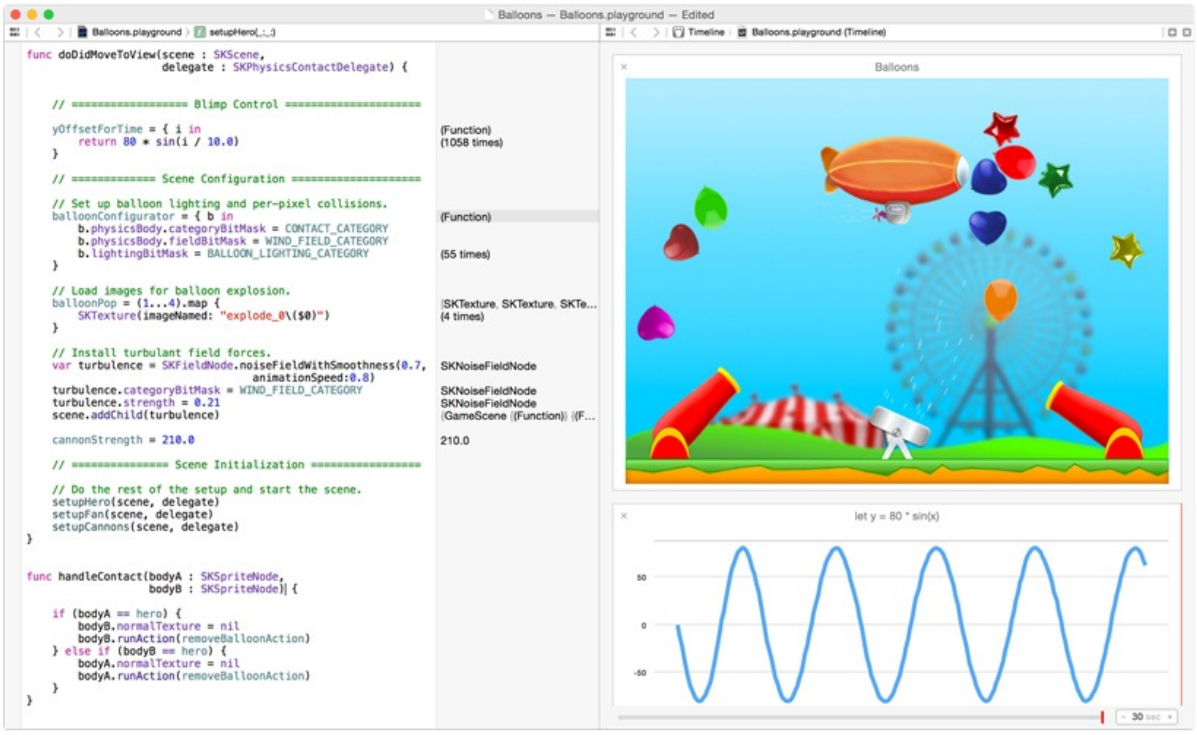
A 3.4 Uniwersum programowania (bardzo)wysokopoziomowego.

2 czerwca 2014 roku podczas swojej największej dorocznej imprezy Worldwide Developers Conference firma Apple zaprezentowała nowy język programowania: Swift przeznaczony do tworzenia softwaru na platformy tego producenta (systemy operacyjne MacOS oraz iOS). Dotychczas tworząc programy na urządzenia firmy z Cupertino należało korzystać z dostarczonego przez nią wariantu języka Objective-C.

Kiedy założona w 1985 przez odsuniętego od Apple Stevena Jobsa roku firma NeXT w drugiej połowie lat dziewięćdziesiątych ubiegłego stulecia została wykupiona przez przedsiębiorstwo z jabłkiem w logo - poza osobą swojego CEO wprowadziła ona do nowej firmy-matki swoje oprogramowanie (system operacyjny NeXTSTEP), które stało się potem zalążkiem dla iOS'a i MacOS X. Było ono zaprogramowane w oparciu o rozwiązania charakterystyczne dla paradygmatu obiektowego (o którym szerzej w następnym rozdziale) przy użyciu nowoczesnego na ówczesne czasy języka objective-C stanowiącego rozszerzenie klasycznego i doskonale znanego C o wzorowane na Smaltalku właściwości obiektowe. Jednak okazało się, że niewiele dużych firm i organizacji również adaptowało ten język jako swoje główne narzędzie. Z punktu widzenia Apple nie było to zbyt wielkim problemem ze względu na wysoką spójność całego ekosystemu firmy (własne systemy + autorskie platformy + samodzielnie rozwijane narzędzia deweloperskie). Z perspektywy programistów natomiast rzecz nie była najwygodniejsza, szczególnie jeśli ktoś nie specjalizował się wyłącznie w oprogramowaniu dla Apple. Spójność i konsekwencja w budowaniu produktu pozwoliła jednak na utrzymanie się objective-C jako języka przez bardzo długi czas pomimo jego malejącej

atrakcyjności. Czasy jednak uległy zmianie, a wraz z nimi podejście twórców oprogramowania do narzędzi i wymagania wobec nich stawiane. Mechanizm tych zmian jest dla nas szczególnie interesujący z perspektywy refleksji z zakresu studiów nad oprogramowaniem.

Swift generalnie jest językiem nabudowanym na dotychczasowych narzędziach developerskich Apple. To znaczy, że z punktu widzenia zaawansowanego programisty, szczególnie doświadczonego w pracy z objective-C wprowadzone zmiany nie będą wydawać się drastyczne. Stanowi on jednak - zgodnie ze współczesnym wektorem zmian w świecie technologii - wyciągnięcie ręki w stronę użytkowników, którzy chcą stać się twórcami. Sami autorzy nazywają Swifta "objective-C bez obciążeń wynikających z C"[42], które osobie niezaangażowanej mogą się wydać trudne i niepotrzebne. Edytor dostarczany w pakiecie Xcode 6 zawiera również narzędzie nazwane *Playground,* które pozwala na dynamiczne testowanie czy wręcz - jak sugeruje nazwa - *zabawę* tworzonymi aplikacjami.



Ilustracja 32: Swiftowy *Playground*

[42] wypowiedź Craiga Federighi szefa działu *Software Engineering* w Apple podczas pierwszej prezentacji Swifta na Apple Special Event WWDC 2014

Chris Lattner, jeden z twórców Swifta napisał na swoim blogu: "Mam nadzieję, że czyniąc programowanie bardziej przystępnym i zabawnym dotrzemy do nowego pokolenia programistów i pomożemy przedefiniować myślenie o informatyce."[43]

Wysokopoziomowość języka stworzonego przez Apple - działanie w wysokich stopniach abstrakcji, przyjazne procesowi myślowemu i twórczemu programisty pozostaje jednak w dalszym ciągu "przyjazna maszynie". Kod źródłowy stworzony w Swifcie kompiluje się i jest realizowany równie szybko (a czasem nawet z większą efektywnością) co generowany przez dotychczasowe rozwiązania. Wydaje się, że to jeden z ważniejszych kierunków ewolucji języków programowania - w stronę zatarcia różnicy między użytkownikiem a programistą. Oczywiście omawiany język nie robi tego całkowicie - aby tworzyć w nim programy nadal potrzebna jest solidna wiedza informatyczna - nie mniej jednak fakt, że jeden z największych "graczy" rynku komputerowego decyduje się na "odhermetyzowanie" swojego podstawowego narzędzia świadczy o kierunku w jakim rozwija się technologia.

Jeszcze wyraźniej kwestia ta unaocznia się w przypadku innego, stosunkowo nowego i innowacyjnego projektu jakim jest Wolfram Language należący do ekosystemu projektów Wolfram Research obejmującego m.in. oprogramowanie Mathematica, Wolfram Programming Cloud, silnik wiedzy (*knowledge engine)* Wolfram Alpha i wiele innych narzędzi. Stephen Wolfram - fizyk i matematyk, autor systemu algebry komputerowej to osoba, która stoi za tym przedsięwzięciem. W swojej kontrowersyjnej książce *A new kind of science* postuluje on modelowanie złożonych systemów obserwowalnych w naturze za pomocą prostych programów komputerowych, które radzą sobie z tą problematyką lepiej niż "zwykła" matematyka. Projekt Wolfram Language stanowi gest w tym kierunku. Skonstruowany w kierowanej przez fizyka grupie badawczej system jest niezwykły pod wieloma względami - opracowano go tak aby obejmował jak najszerszy zakres informacyjnej rzeczywistości (jest wieloparadygmatowy, wieloplatformowy, pozwala na włączanie gigantycznych baz wiedzy, algorytmów itd.) oraz by funkcjonował na tak wysokim poziome abstrakcji jak to tylko możliwe. Oznacza to, że jego składnia stara się naśladować język naturalny, a sam kod na poziomie programistycznym w żaden sposób nie odnosi się do hardwaru, na którym ma być realizowany. Prezentowany na stronie

[43] http://www.nondot.org/sabre/ (dostęp 1.07.2014)

internetowej projektu[44] przegląd właściwości języka (*principles and concepts)* wymienia cechy, które w większości są unikatowe dla Wolfram Language:

- *knowledge based programming* zakłada wykorzystanie tak dużej bazy wiedzy informatycznej jak to tylko możliwe, środowisko WA posiada np. wbudowaną imponującą bazę gotowych algorytmów;
- *automatyzacja*, łączy się z poprzednią kwestią: chodzi o to, żeby korzystać z jak najszerszego repozytorium gotowych rozwiązań;
- *koherencja* oznacza, że pisząc kod zgodny ze specyfikacją nie musimy zwracać uwagi na zgodność typów czy współpracę różnych konstrukcji programistycznych; jeśli napisaliśmy poprawny kod w Wolframie - będzie on działał;
- cały język jest zbudowany na *wyrażeniach symbolicznych* - dane, formuły, kod, grafika, dokumenty: każdy z tych obiektów ma w nim tego rodzaju reprezentację co pozwala na elastyczne manipulowanie zarówno strukturami jak i zawartością;
- jedną z najbardziej interesujących kwestii jest fakt, że w omawiany system pozwala odnosić się kodem do *modelu świata rzeczywistego* w bardzo intuicyjny sposób; możemy używać danych geolokacyjnych, jednostek miar i wag, a także wzorców opisujących miliony konkretnych *przedmiotów* (!);
- Wolfram Language w sposób niespotykany pozwala na wprowadzenie *wyrażeń języka naturalnego* do kodu źródłowego (!), nie tylko na zasadzie pojedynczych wyrażeń-kluczy (typu polecenie "write");

Wziąwszy pod uwagę wymienione powyżej cechy, możemy bezpiecznie nazwać Wolfram Language językiem bardzo wysokiego poziomu - samo twórcy jako jedno z jego zastosowań podają *ultra-high level scripting*. Przykładowy kod w nim realizowany może wyglądać następująco[45]:

```
gameOfLife = {224, {2, {{2, 2, 2}, {2, 1, 2}, {2, 2, 2}}}, {1, 1}};
board = RandomInteger[1, {50, 50}];
Dynamic[ArrayPlot[board = Last[CellularAutomaton[gameOfLife, board, {{0, 1}}]]]]
```

wygeneruje nam, znany już z poprzednich rozdziałów automat *Gry w Życie*. Pierwsza linia kodu implementuje reguły obowiązujące w conwayowskim układzie, druga generuje siatkę

[44] http://www.wolfram.com/language/principles/ (dostęp 1.08.2014)

[45] źródło: dokumentacja na stronie http://www.wolfram.com/language (dostęp 1.08.2014)

50x50 wypełniając ją losowymi liczbami naturalnymi z zakresu 0 do 1 (czyli zerami i jedynkami), trzecia "rysuje" dynamiczny układ. Jak widać program, którego implementacja w Processingu zajmuje co najmniej kilkadziesiąt linii kodu tutaj zredukowana została do trzech, co daje wyobrażenie na temat siły ekspresyjnej języka i czyni go niezwykle ciekawym narzędziem do szybkiego prototypowania. Wolfram Language nadaje się również wyśmienicie do przetwarzania i wizualizacji informacji zawartych w różnego rodzaju źródłach (dokumentach, bazach danych, obrazach etc).

Co ciekawe pomimo dużych możliwości WL nie należy do najpopularniejszych języków programowania. Prawdopodobnie wynika to z faktu, że świat IT upodobał sobie specjalizację i nadmierna elastyczność - tak jak w przypadku Lispu - może początkowo wydawać się czymś nie do końca pożądanym. Duzi producenci oprogramowania ze silną nieufnością spoglądają na projekty, które wprowadzają potężne, rewolucyjne wręcz zmiany w bardzo krótkim czasie - pojawiają się pytania o stabilność i skalowalność tego rodzaju rozwiązań. Dlatego takie firmy jak Apple wolą na razie rozwijać swoje narzędzia ostrożniej - tak jak to miało miejsce z wprowadzeniem Swifta, który został zaproponowany jako standard równolegle z objective-C, a nie zamiast niego. Daje to wyspecjalizowanym programistom czas na zapoznanie się z nowym zjawiskiem, nie odcinając ich od aktualnie zdobytego doświadczenia - nowi twórcy oprogramowania zaś, zgodnie z przewidywaniami, zaczną naukę od razu od nowego języka, wchodząc od razu do uniwersum wyższego poziomu. Wolfram Language, ponieważ nie jest związany z żadną fizyczną platformą ani nie "ciągnie za sobą balastu" w postaci już zaimplementowanych rozwiązań biznesowych, które muszą być wspierane dla ciągłości i spójności ekosystemu - może być futurystycznym skokiem naprzód, do którego reszta świata być może za jakiś czas dołączy. Tak czy inaczej wydaje się, że programowanie (bardzo) wysokopoziomowe to przyszłość rozwoju technologii informatycznych. Jest bardzo prawdopodobne, że w niedługim czasie wszyscy będziemy oprogramowywać nasze wszechobecne urządzenia komputerowe (w postaci samochodów, inteligentnych domów) stosując dostosowane do wiedzy informatycznej zwykłego użytkownika "klocki", będące tak naprawdę gotowymi fragmentami kodu i algorytmami.

Αναλυτικά 4: Paradygmaty programowania.
Od kodu maszynowego do obiektowych ontologii.

"PARADYGMAT (παράδειγμα) - 1) W języku metafizyki Platona termin ten oznacza «model», wg którego są zbudowane rzeczy zmysłowe, wyraża zatem normatywność ontologiczną idei (II, 100 nn.). Trzeba zwrócić uwagę na to, że Platon mówi także o «paradygmacie życia», to znaczy o modelu, który dusza wybiera, zanim przyjdzie na świat. Chce przez to powiedzieć, że każdy typ życia ma bardzo ściśle określone znaczenie metafizyczne (II, 236 nn.). - W ujęciu Filona idee przestają być paradygmatami absolutnymi a stają się tylko paradygmatami względnymi, to znaczy obrazami w stosunku do Boga (który jest modelem absolutnym), a paradygmatami w stosunku do rzeczy; (...) W logice Arystotelesowskiej paradygmat oznacza przykład (...)" (Reale, 2005, pp. 166-167)

"Wybór między paradygmatami jest wyborem między nie dającymi się pogodzić sposobami życia społecznego. Ze względu na swój charakter wybór ten nie jest i nie może być zdeterminowany wyłącznie przez metody rozstrzygania problemów w nauce instytucjonalnej, te bowiem zależą częściowo od określonego paradygmatu, który właśnie jest zakwestionowany. Z chwilą gdy w sporze o wybór paradygmatu odwołujemy się do paradygmatu - a innego wyjścia nie mamy - popadamy w nieuchronnie w błędne koło" (Kuhn, 1968, p. 111)

"Programowanie to tworzenie oprogramowania w celu rozwiązania problemu (...) Zamierzamy zastosować pojęcie paradygmatu w badaniu języków programowania i architektur oprogramowania aby ustalić jak skutecznie jesteśmy w stanie rozwiązywać różnego rodzaju problemy." (Kaisler, 2005, pp. 1,3)

"Projektowanie oznacza tworzenie rzeczy dla ludzi. Ale mówiąc o ludziach nie mamy na myśli tylko użytkowników. Projektant też jest człowiekiem" (Graham, 2004, p. 258)

Podobnie jak w wielu innych dziedzinach wiedzy, w tworzeniu oprogramowania obowiązują - poza regułami charakterystycznymi dla składni i semantyki konkretnego języka - style i podejścia w rozumieniu tego jak jest zbudowany i jak ma działać złożony kod komputerowy. Dotychczas przedmiotem naszego zainteresowania była głównie logika, teoria informacji i kodowania, co prowadziło w stronę refleksji nad podstawowymi mechanizmami komputacji i egzekucji poleceń w systemach elektronicznych.

Przyjrzeliśmy się też budowie programów (czyli algorytmów i struktur danych ubranych w składnię języka), a także ich specjalnym przypadkom aspirującym do miana form artystycznej ekspresji. Jednak programy to nie tylko wykonywany krok po kroku kod źródłowy wysokiego lub niskiego poziomu - w przypadku współczesnych fenomenów tego rodzaju mamy do czynienia ze strukturami o wysokiej złożoności, które analizować można na różnych poziomach abstrakcji, włączając w obraz całości relacje między nimi. Okazuje się bowiem, że języki programowania mogą operować w ramach zróżnicowanych ontologii i stylistyk opracowanych na potrzeby optymalizacji rozwiązywania zadanego problemu.

A 4.1. Programowanie imperatywne.

Jednym z bardzo ważnych historycznie konceptów rozwijających ideę Alana Turinga dotyczącą skonstruowania maszyny manipulującej znakami jest architektura von Neumanna prezentowana już na marginesie rozważań rozdziału A 1.2. Charakterystyczną cechą tego rozwiązania jest fakt wspólnego przechowywania danych z instrukcjami w adresowanej pamięci. Ponieważ architektura Princeton znalazła zastosowanie we wczesnych komputerach takich jak angielski EDSAC czy amerykański EDVAC wraz z późniejszymi rozwinięciami - sposób funkcjonowania cykli typu *fetch-execute* i będący jego rezultatem sposób przechowywania zmiennych znalazł odzwierciedlenie w postaci *imperatywnych* języków programowania.

Paradygmat imperatywny polega, najprościej rzecz ujmując na tym, że programista operuje na stanach i komendach modyfikujących te stany. Jeśli jedne tego typu konstrukcje zostaną włączone w inne jako podprogramy to będziemy mieli do czynienia z programowaniem *proceduralnym.*

$$S_0 \times Op_0 \rightarrow S_1 \times Op_1 \rightarrow S_2 \times Op_2 \rightarrow \cdots S_{n-1} \times Op_{n-1} \rightarrow S_n$$

Schemat przejść między stanami (za: Kaisler, 2005, p. 23)

Z punktu widzenia hardwaru stan jest zawartością pewnego miejsca w pamięci, natomiast instrukcje przekładają się na komendy kodu maszynowego czyli manipulacje zawartością pamięci. Stąd programowania w paradygmacie imperatywnym nie umiejscawiamy bardzo wysoko w rankingu abstrakcji. Nie mniej jednak konstrukcje w nim obecne takie jak pętle warunkowe, zagnieżdżanie procedur oraz w większości przypadków

wygodna i przejrzysta składnia powodują, że pisanie kodu w tym paradygmacie jest znacznie wygodniejsze od programowania w asemblerach. Do najpopularniejszych języków silnie osadzonych w proceduralnym sposobie rozwiązywania problemów obliczeniowych należą C czy dawniejsze BASIC lub Pascal, aczkolwiek należy zaznaczyć ten sposób realizowania kodu daje się stosować również w przypadku języków, które naturalnie wspierają paradygmat obiektowy - np. znanym nam już Processingu.

Przykładowy kod proceduralny może wyglądać następująco (jest to zagnieżdżona instrukcja if-else w języku C):

```
main()
{
int x;
printf("\n Liczba: ");
scanf("%d",&x);
     if(x%2==0)
          if(x>0)
          printf("\n %d jest parzysta i dodatnia.",x);
          else
          printf("\n %d jest parzysta i ujemna.",x);
     else
          if(x>0)
          printf("\n %d jest nieparzysta i dodatnia.",x);
          else
          printf("\n %d jest nieparzysta i ujemna.",x);
getch();
return 0;
}
```

Program pobiera liczbę i w instrukcji warunkowej sprawdza jej parzystość i dla obu możliwych wyników w kolejnej instrukcji warunkowej sprawdza czy jest ona większa czy mniejsza od zera wypisując odpowiedni wynik. Następnie czeka na naciśnięcie dowolnego klawisza (polecenie `getch()`).

A 4.2 Programowanie deklaratywne.

W opozycji do programowania imperatywnego stoi rodzina *deklaratywnych* styli programowania. Podstawowa różnica leży tu przede wszystkim w sposobie konceptualizowania problemów. W poprzednio omawianym wypadku aby rozwiązać zadanie należało zaimplementować krok po kroku algorytm prowadzący do jego rozstrzygnięcia, używając ewentualnie konstrukcji w rodzaju pętli itp. - tutaj rzecz polega raczej na postawieniu hipotezy na temat pożądanego do uzyskania wyniku i przeprowadzeniu swoistej procedury sprawdzającej. Peter van Roy oraz Serif Haridi wymieniają listę kryteriów dla programów deklaratywnych: *niezależność* od stanów zewnętrznych, *bezstanowość i detrminizm (Peter van Roy, 2004)*. Języki tego rodzaju są w sposób konieczny wysokopoziomowe ze względu na fakt, że opisują logikę tego co ma zostać wykonane bez uwzględniania dokładnie *jak* ma to być zrealizowane przez konkretną maszynę. Przy takim podejściu dane mogą być opisywane jako proste fakty ale również jako zasady lub zespoły zasad opisujących fakty.

Jednym z rodzajów programowania deklaratywnego jest programowanie *funkcyjne ,* w którym kod - jak sama nazwa wskazuje - składa się wyłącznie z funkcji. Jest nią również sam program, który przyjmuje dane wejściowe jako swoje argumenty i dostarcza danych wyjściowych jako wynik. Jednym z najlepszych przykładów funkcyjnych języków programowania jest Haskell, zaś do innych należą omawiany już wcześniej Lisp (choć są w tym względzie pewne kontrowersje), SML czy CAML. Tak naprawdę nie istnieje silna definicja języków tego rodzaju:

"Programowanie funkcyjne to taki styl programowania, który kładzie nacisk na ewaluację wyrażeń raczej niż wykonywanie poleceń. Wyrażenia w tych językach są formułowane za pomocą funkcji łączących podstawowe wartości. Język funkcyjny to taki, *który wspiera i zachęca do pracy w funkcyjnym stylu.* (podkreślenie J.K.A.)" (Kaisler, 2005, p. 24)

w tym kontekście wydaje się, że skutecznie da się wskazać raczej jedynie języki, które z całą pewnością do tej grupy nie należą niż te, które są w sposób konieczny funkcyjne.

Innym rodzajem języków deklaratywnych są języki *logiczne* takie jak opisywany już Prolog. Tego rodzaju narzędzia dostarczają konstrukcji pozwalających na definiowanie

faktów oraz przeprowadzanie na nich *implikacji.* Działanie programu polega na przeprowadzeniu *rozumowania* w sensie logicznym na zbiorze faktów za pomocą zbioru reguł i próbie określenia prawdziwości zadanych wypowiedzi. Języki tego typu świetnie nadają się do budowania dynamicznych baz wiedzy, sztucznej inteligencji czy systemów ekspertowych.

A 4.2 Paradygmat obiektowy.

Programowanie obiektowe jest dziś dominującym paradygmatem tworzenia oprogramowania choć stworzone zostało stosunkowo dawno - za pierwszy język prawdziwie obiektowy uważana jest Simula 67 Ole-Johana Dahla i Kristena Nygaarda (numer przy nazwie oznacza datę dokonanej przez nich rewizji języka Simula I - 1967 rok). Zaproponowana przez badaczy koncepcja okazała się niezwykle trwała, przede wszystkim ze względu na fakt, że doskonale sprawdza się ona w przypadku projektów o dużej i bardzo dużej złożoności.

W programowaniu proceduralnym mieliśmy do czynienia z sytuacją gdzie dane i procedury są od siebie oddzielone. Podstawą ujęcia obiektowego jest potraktowanie danych i zachowań (działań na nich wykonywanych) jako pewnej całości (czyli: obiektu). Najłatwiej wyjaśnić działanie paradygmatu wyliczając powszechnie uznawaną listę jego właściwości (np. za: Weisfeld, 2010):

Abstrakcja - obiekt w sensie programistycznym jest modelem aktywnego *bytu*, który może wchodzić w relację z innymi obiektami (komunikować się, zmieniać swój stan, opisywać). Te autonomiczne fragmenty kodu mogą służyć za wzorce dla innych fragmentów. W ten sposób możliwe jest zdefiniowanie swego rodzaju "szablonu", z którego mogą lecz nie muszą być generowane konkretne instancje. Co więcej: pewne klasy stanowią wzorce dla innych klas, które powstają na ich podstawie, samemu nie służąc do tworzenia obiektów.

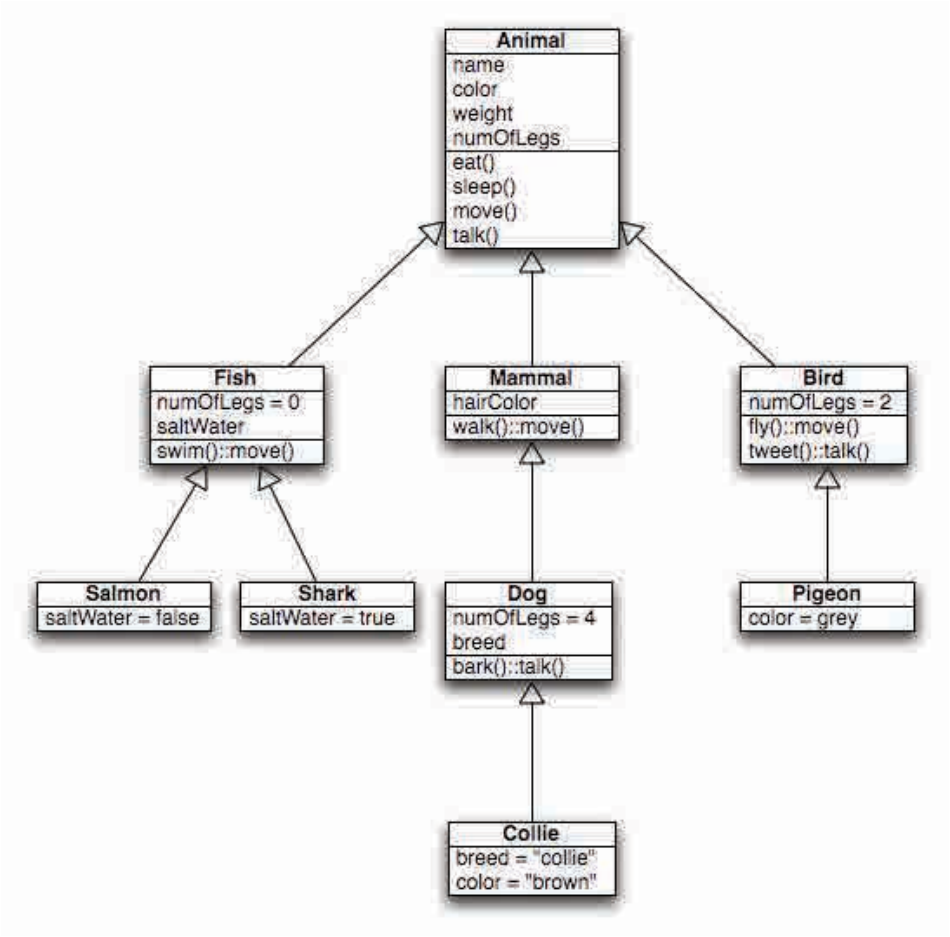
Hermetyzacja / enkapsulacja / obfuskacja - proces polegający na blokowaniu dostępu do stanów wewnętrznych między obiektami. Idea polega na tym, że obiekt udostępnia pozostałym jedynie swój interfejs czyli listę sposobów na jakie mogą one wejść z nim w interakcję. Dzięki temu programista może być pewny, że na danym obiekcie nie da się przeprowadzić żadnych niespodziewanych operacji. Dodatkowo w przypadku współpracy wielu osób nad jednym projektem współprogramiści nie muszą wiedzieć jak dany obiekt

realizuje swoje zadania lecz jedynie jak się z nim komunikować (czyli jakiego rodzaju dane przyjmie i jaki zwróci wynik).

Na interesujące konsekwencje tego faktu zwraca uwagę Aleksander R. Galloway w swoim artykule *Język chce, aby go nie dostrzegać:*

"Fundamentalną zasadą działania zorientowanych obiektowo języków komputerowych jest połączenie zmiennych i operacji na zmiennych (metod) w postaci tak zwanej klasy. Możliwe jest tworzenie wielu instancji klas w postaci obiektów, zaś obiekty te komunikują się ze sobą za pomocą interfejsów. Tu właśnie wkracza enkapsulacja: szczegóły dotyczące tego, w jaki sposób obiekt implementuje daną operację są celowo ukryte przed każdym innym żądającym tej operacji obiektem (...) Ukrywanie informacji stosuje się, by kod był bardziej modułowy i abstrakcyjny, a więc łatwiejszy w obsłudze (...) Być może zabrzmi to zbyt dosłownie, ale warto podkreślić istotne podobieństwo pomiędzy takim opisem procesu pracy a tym samym opisem przedstawionym przez Marksa i jego następców w dyskusji o ideologii. W obu przypadkach mamy do czynienia z »obiektem« obdarzonym złożoną maszynerią służącą ukrywaniu rzeczy, czy towaru (...) i jego zdolności do maskowania własnej historii produkcji i generującego ją społecznego podziału pracy (...) Kod jest medium, które nie jest medium. Nigdy nie oglądamy go takim, jakie jest, a jedynie w postaci skomplikowanej, zinterpretowanej, odczytanej i na inne sposoby sprowadzonej do ukrycia przez jeszcze większe zlepki kodu. Stąd zasada *ukrywania."* (Aleksander Galloway, 2008, pp. 29-30)

Dziedziczenie - to jedna z najważniejszych cech paradygmatu pozwalająca na daleko bardziej wyrafinowane korzystanie z napisanego kodu (*reuse)* niż znane nam już wielokrotne używanie tych samych procedur. W ujęciu obiektowym mamy możliwość skonstruowania powiązań między klasami tak, że jedna z nich przejmie atrybuty i metody innej. Nadklasa (klasa nadrzędna) zawiera atrybuty i zachowania, które stanowią część wspólną wszystkich dziedziczących po niej, tak jak to widać na poniższym diagramie:



Ilustracja 33: Diagram dziedziczenia klas - widzimy,
że podklasy zawierają wszystkie właściwości klas nadrzędnych.[46]

Polimorfizm - oznacza możliwość abstrakcji wyrażeń od konkretnych typów. Matt Weisfeld tłumaczy działanie tego mechanizmu następująco:

"Jeśli do obiektu zostanie wysłany komunikat, obiekt musi mieć odpowiednią metodę aby móc odpowiedzieć. W hierarchii dziedziczenia podklasy przyjmują interfejsy po swoich nadklasach. Jednak ze względu na to, że każda klasa jest osobnym bytem, może wymagać innego rodzaju reakcji na ten sam komunikat (...) Mówiąc krótko, każda klasa może w inny sposób zareagować na wywołanie (tej samej -dop. J.K.A.) metody ..."

(Weisfeld, 2010, p. 41)

Aby lepiej zrozumieć konstrukcję programów w ramach paradygmatu obiektowego przyjrzyjmy się przykładowi[47] kodu opisującego hipotetycznego "kota" w "biegu". Jest to zwykła, proceduralna aplikacja napisana w Processingu:

---

[46] źródło: http://www.silentlycrashing.net/p5/objects_3.php (dostęp 1.08.2014)

[47] za dokumentacją na stronie http://www.processing.org/tutorials/objects/ (dostęp 1.08.2014)

```
color c;
float xpos;
float ypos;
float x speed;  // deklarujemy zmienne: kolor, położenie i prędkość kota

void setup() {
      size (200,200);
      c = color(255);   // w wykonywanej jednokrotnie na starcie programu pętli
      xpos = width/2;   // przypisujemy wartości zmiennym
      ypos = height/2;
      xspeed = 1;  }

void draw() {
      background(0);    // w pętli draw wywołujemy procedury odpowiedzialne za
      display();        // narysowanie kota i jego przemieszczanie
      run(); }          // procedury są opisane poniżej

void display() {
      rectMode(CENTER);
      fill(c);
      rect(xpos, ypos, 20, 10); } // nasz kot okazuje się prostokątem

void run() {
      xpos = xpos + xspeed;
      if (xpos > width) {
      xpos = 0;               // animacja polega na przesuwaniu kota w osi x
      } }                     // o wartość równą xspeed w każdej klatce
```

przepisując całość w paradygmacie obiektowym otrzymamy *podobny* kod podstawowy:

```
class Kot                     // definiujemy klasę - wzorzec
      color c;
      float xpos;
      float ypos;
      float xspeed;            // dane jakie będą nam potrzebne

Kot() {                       // klasa zawiera opis konstruktora
      c = color(255);
      xpos = width/2;
      ypos = height/2;
      xspeed = 1;
}
```

Pozostała część źródła będzie wyglądała tak samo: tworzymy procedury `display()` i `run()`. Zupełnie inaczej będzie jednak przebiegało uruchomienie programu. Na bazie powyższego kodu deklarujemy obiekt:

```
Kot mojKotBlekot;
```

i inicjujemy go w pętli setup:

```
mojKotBlekot = new Kot();
```

a następnie wywołujemy metody obiektu w pętli draw:

```
mojKotBlekot.run();
mojKotBlekot.display();
```

co ważne, możemy teraz bardzo łatwo powołać kolejne instancje naszego obiektu i je parametryzować (o ile zadbaliśmy o taką możliwość w konstruktorze):

```
Kot mojKotBlekot = new Kot();
Kot mojKotHermia = new Kot();
```

itd.

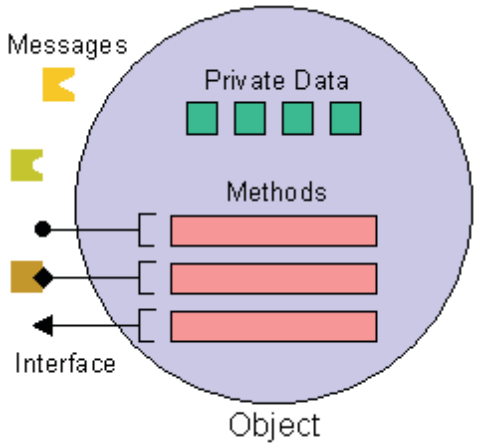
Skoro przyjrzeliśmy się najprostszemu przykładowi programu napisanego w paradygmacie obiektowym możemy usystematyzować pojęcia:

Obiekt to "samodzielny" blok kodu składający się z danych i zachowań.

Klasa jest szablonem do tworzenia obiektów.

Metoda to po podprogram związany z klasą obiektu, który definiuje jego zachowanie.

Interfejs zbiór metod, które specyfikują jak można komunikować się z obiektem.



Ilustracja 34: budowa obiektu i działanie interfejsu[48]

Z zaprezentowanej ogólnej systematyki wynika, że w przeciwieństwie do poprzednio analizowanych modeli tworzenia oprogramowania - paradygmat obiektowy oferuje twórcom swego rodzaju ontologię, w której kod opisywany jest w kategoriach podobnych do rzeczy w świecie - jest pewną ograniczoną strukturą, posiadającą swoje właściwości, o której możemy orzekać pewne prawdy i z którą możemy się komunikować. Ontologia obiektowa wynika z konkretnej, funkcjonalnej i pragmatycznej wizji świata gdzie procesy i zjawiska daje się modelować do postaci różnego rodzaju abstrakcyjnych relacji (bardzo statycznych w swojej powtarzalności).

Paradygmat obiektowy jest obecnie najbardziej powszechnym sposobem konstruowania softwaru, szczególnie w dużych zastosowaniach komercyjnych. Dzieje się tak ze względu na obecność licznych mechanizmów ułatwiających zbiorową pracę na dużych modułach kodu. Pracę w OOP (*Object Oriented Paradigm)* umożliwiają praktycznie wszystkie największe i najbardziej popularne języki programowania: C++, Java, Common Lisp, JavaScript, objective-C, C#, Perl, PHP, Ruby, .NET i wiele innych. Z punktu widzenia *studiów nad oprogramowaniem* najciekawszy wydaje się problem czy fakt przyjęcia takiej a nie innej metodologii tworzenia softwaru przekłada się w jakiś sposób na sposoby produkcji, dystrybucji i konsumpcji danych kulturowych. Wstępnie przyjrzymy się tej kwestii w kolejnym podrozdziale, a następnie będziemy kontynuować elementy tej refleksji w trzeciej części pracy, traktującej o dynamikach związanych z powszechną augmentacją naszej rzeczywistości przez świat technologii komputerowych i informacyjnych.

[48] źródło: http://courses.cs.vt.edu/csonline/SE/Lessons/OOP/object.gif (dostęp: 1.08.2014)

Αναλυτικά 5: Critical Code Studies.

"Zamiast definiować *tekst* jako łańcuch elementów znaczących, jak to robią lingwiści i semiotycy, używam tego słowa w odniesieniu do całego zakresu fenomenów - od krótkich wierszy po złożone programy komputerowe i bazy danych. Jak wskazuje przedrostek *cyber* (w "*cybertekście"* - dop. J.K.A.) tekst jest postrzegany jako maszyna - nie metaforycznie ale jako mechaniczne urządzenie dla produkcji i konsumpcji słownych znaków" (Aarseth, 1997, pp. 20-21)

"W swoim pierwszym sensie kod jako *tekst* jest produktem praktyki programowania lub kodowania. Najczęściej przyjmuje formę zbioru plików tekstowych zawierających kod źródłowy lub skrypt (...). Kodowanie, programowanie i hakowanie wytwarzają kod. Kod w tym sensie jest zawsze napisany w jakimś języku - C, Perlu, Javie, asemblerze, Lispie, Cobolu, Basicu, etc. Praktyki czytania i pisania kodu wahają się od wysoce kreatywnych i klarownych po zagmatwane i powtarzalne (..) Zarówno wzajemna rywalizacja jak i merytoryczna dyskusja przesuwają nieustannie znaczenie i wartość pracy w kodzie. Niekiedy spory odnoszą się do »stylu« i rzemieślniczo-estetycznych sądów na temat tego jak kod powinien być formatowany na ekranie, jak należy nazywać zmienne, etc. Czasami kłótnie mają szersze wymiary ideologiczne." (Mackenzie, 2003a, pp. 5-6)

"Oprogramowanie działa poprzez technologiczny model, który premiuje żmudne deklaracje symboliczne i opisy, choć jednocześnie wymaga ukrywania, enkapsulacji i zaciemniania wielkich fragmentów kodu. Twierdzenie brzmi następująco: *oprogramowanie potrzebuje zarówno refleksji jak i obfuskacji (zaciemniania).* Jeżeli oprogramowanie jest faktycznie alegoryczną analogią ideologii, nie powinno być zaskoczeniem, że działa ono w taki dialektyczny sposób. Krytycy ideologii opisywali ją często w sposób syntetyczny (Jameson, Hall, Gramsci i inni). Ale oprogramowanie wyposażone jest we własne technologie refleksji i ukrywania. Refleksja jest niemalże aksjomatyczna: komplet reguł składniowych i znaczeniowych języka komputerowego musi być zdefiniowany i wpisany w każde środowisko zaprojektowane do interpretowania, odczytywania lub wykonywania tego języka (...) zaciemnianie kodu, leży również u podstaw tego, jak tworzone jest oprogramowanie." (Aleksander Galloway, 2008, p. 28)

Studia krytyczne nad oprogramowaniem to coraz bardziej dynamicznie rozwijające się pole refleksji akademickiej. Do niedawna kwestia badań nad sposobami działania języków programowania, ich wpływu na funkcjonowanie jednostek i społeczeństw, a także na obiegi komunikacji medialnej zarezerwowana była niemal wyłącznie dla informatki i jej działów, ewentualnie nauk pokrewnych. Nie licząc odosobnionych przypadków (np. refleksji Friedricha Kittlera, Nicholasa Negroponte czy Espena Aarsetha) refleksja humanistyczna i kulturoznawcza koncentrowała się raczej na rezultatach działania technologii obliczeniowych, ewentualnie tych aspektach ich funkcjonowania, które mogły być bezpośrednio doświadczane przez użytkownika (np. interfejsach). Ta perspektywa uległa istotnej zmianie przede wszystkim w ciągu pierwszej dekady XXI wieku. Badania koncentrujące się na estetyce "powierzchni ekranu" i treściach komunikowanych przez media komputerowe mniej lub bardziej wprost zostały poszerzone o próby analizy wewnętrznych, "niewidzialnych" mechanizmów działania oprogramowania nimi zawiadującego. Do pionierów takiego podejścia zaliczyć można wymienionego już F. Kittlera, a także Davida Berry, Mathew Fullera, Wendy Chun, Katherine Hyles, Nicka Montforta, Iana Bogosta czy Lwa Manovicha.

Do podstawowych strategii analizy oprogramowania z perspektywy humanistycznej należeć mogą: przyjrzenie się kodowi źródłowemu (z uwzględnieniem języka programowania), warstwie koncepcyjnej softwaru (architektura oprogramowania) czy sposobom przetwarzania danych przez systemy informatyczne i konsekwencjom ich działania (dynamiki, polityki i ideologie kodu).

Kod jako tekst

Software jest jednocześnie tekstem i procesem. Kod jako tekst jest rezultatem praktyki programowania ujętej w bardzo ścisłe reguły. We wstępie do jednego z podręczników skierowanych do programistów czytamy:

"Czy zdarzyło ci się kiedykolwiek...

zmarnować dużo czasu na pisanie niewłaściwego algorytmu?

użyć zbyt skomplikowanej struktury danych?

pominąć oczywisty błąd w testowanym programie?

spędzić cały dzień na szukaniu takiego błędu?

przerabiać program, aby działał trzy razy szybciej i zużywał mniej pamięci?

przenosić program, ze stacji roboczej na komputer PC i odwrotnie?

próbować wprowadzić sensowne zmiany w programie napisanym przez kogoś innego?

przepisać program od nowa, bo nie dało się go zrozumieć?

(...)

Podobne rzeczy zdarzają się programistom nieustannie ale nie zawsze można sobie łatwo z nimi poradzić. Główną przyczyną tego problemu jest to, że takie zagadnienia, jak testowanie, diagnostyka, przenośność, wydajność, alternatywy projektowe i styl - **praktyka** programowania - są często na zajęciach z informatyki i programowania traktowane po macoszemu." (Brian W. Keringhan, 2011, p. 7)

Powyższe wyliczenie możemy wprost potraktować jako listę zainteresowań swoiście rozumianej *stylistyki komputerowej.* Założenie byłoby następujące: otrzymujemy do analizy działający (dający się skompilować lub uruchomić w odpowiednim środowisku) kod. Następnie:

Możemy przyjrzeć się jego konstrukcji z punktu widzenia estetyki - jego czytelności (dla człowieka), zwięzłości, brakowi/obecności nadmiernej redundancji, nazewnictwu zmiennych, komentarzom, skalowalności, uniwersalności, przenoszalności.

Naszą uwagę możemy też poświęcić kwestii adekwatności kodu i zadania, które ma on realizować - czy czyni to z odpowiednią wydajnością, czy jest optymalny, czy nie zawiera oczywistych błędów (kod działa - to wiemy z założenia; analiza całkiem niedziałającego kodu ma sens wyłącznie w szczególnych wypadkach - decydujący jest tu kontekst; co innego *źle* działający kod - taki może być bardzo interesujący).

Ostatnim polem zainteresowania analizy tekstualnej kodu byłaby jego innowacyjność, oryginalność rozwiązań, *twórczy charakter.*

Warto formułując postulat badań tekstualnych nad kodem zwrócić uwagę na trzy nietypowe przypadki "oprogramowania" czyli struktury, które wyglądają jak kod ale nie działają (niektóre przypadki *code artu*); programy, których kod źródłowy nie wygląda jak kod ale te z kolei dają się kompilować (języki *ezoteryczne*) oraz całkowicie poprawnie napisane programy stworzone w klasycznych językach, które jednak nie realizują żadnej funkcji pragmatycznej - są pomyślane jako wypowiedź artystyczna/poezja.

Przykład: Poezja, która przypomina *kod źródłowy* (ale nim nie jest)

Tego rodzaju przypadki są stosunkowo rzadko spotykane. Generalnie fakt niewpisywania się takiego *źródła* w struktury żadnego działającego kompilatora czy środowiska uruchomieniowego dyskwalifikuje ten rodzaj tekstu jako software. Jedynym przypadkiem byłaby tu poezja pseudokodu - taka jak "przepis na Algorystę" przytoczony w jednym z wcześniejszych rozdziałów. Pseudokod należy do świata informatyki lecz pełni funkcję wyjaśniającą lub dokumentacyjnią - nie stanowi programu w ścisłym sensie.

Przykładem omawianego przypadku może być utwór autorstwa Grety Kikilaite *Lean Guy*, który jest syntaktycznie poprawnym kodem źródłowym lecz nie da się go skompilować bez dodatkowej wiedzy na temat zawartości zmiennych w rodzaju "`ago_lived_there_a_guy`":

```
public class lean_guy {
public static void main(String[] args) {

long ago_lived_there_a_guy;
float ed_he_accross_the_sky;
boolean called_him_all_his_friends,
couse_he_was_as_lean_as_hell;

char ming_eyes_he_had_as_matched,
but_the; String was_still_attached;
final float he_did_today;
long before_he_flew_away;
}
}

/* (C) Greta Kikilaite 2013 */
```
[49]

Przykład: Języki *ezoteryczne*

Ezoteryczne języki programowania powstają najczęściej jako żart, wypowiedź artystyczna albo po prostu jako spektakularne przykłady tego, że istnieje w zasadzie nieograniczona ilość możliwości na skonstruowanie systemów, które będą językami kompletnymi w sensie Turinga.

---

[49] źródło: http://instacod.es/66667 (dostęp 3.08.2014)

Do najbardziej popularnych języków ezoterycznych należy *Brainfuck* ze względu na swój minimalizm, który prowadzi do kompletnego braku czytelności kodu. Język ten zawiera tylko osiem znaków: `< > + - . , [ ]` za pomocą których da się realizować dowolne zadania, jakie może wykonać Maszyna Turinga. Kod Brainfucka generujący "Hello world" wygląda następująco:

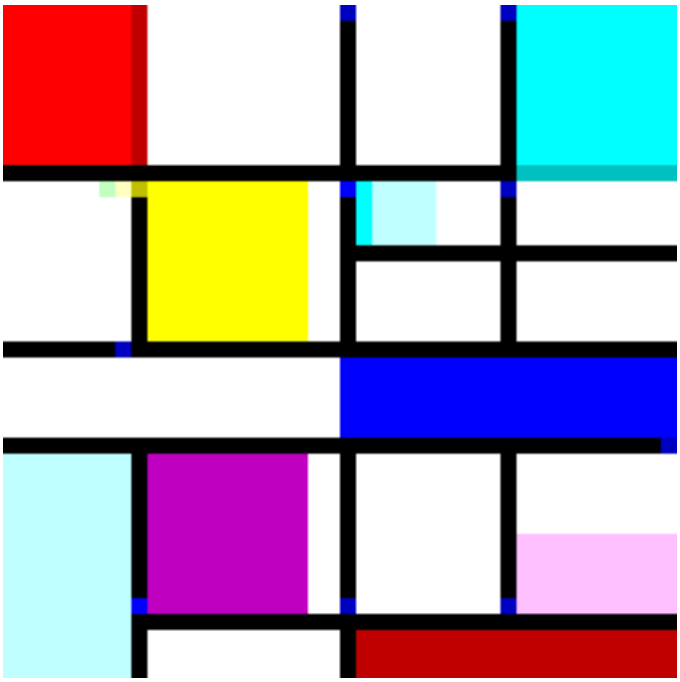
```
++++++++++[>+++++++>++++++++++>+++<<<-]>++.>+.+++++++
  ..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.
```

Ewidentnie żartobliwy charakter ma natomiast język - pochodzący od wyżej opisywanego - o nazwie "Oook!", który w założeniu ma symulować sposób wypowiadania się orangutana-bibliotekarza, jednej z najbardziej charakterystycznych postaci cyklu książek o *Świecie Dysku* Terry'ego Pratchetta. "Hello World" dla tego języka ma postać:

```
Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook! Ook? Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook? Ook! Ook! Ook? Ook! Ook? Ook.
Ook! Ook. Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook! Ook? Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook?
Ook! Ook! Ook? Ook! Ook? Ook. Ook. Ook. Ook! Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook. Ook! Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook! Ook. Ook. Ook? Ook. Ook? Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook? Ook? Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook? Ook! Ook! Ook? Ook! Ook? Ook. Ook! Ook.
Ook. Ook? Ook. Ook? Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook! Ook? Ook? Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook? Ook! Ook! Ook? Ook! Ook? Ook. Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook.
Ook? Ook. Ook? Ook. Ook? Ook. Ook? Ook. Ook! Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook! Ook. Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook.
Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook! Ook!
Ook! Ook. Ook. Ook? Ook. Ook? Ook. Ook. Ook! Ook.
```

Jak widać język ten nie należy do zbyt wydajnych ani czytelnych.

Do znacznie bardziej wyrafinowanych przykładów ezoterycznych języków programowania należą wizualny Piet, którego kod źródłowy przybiera postać bitmap, których kolory i kompozycja mogą być *wykonywane* przez odpowiedni interpreter:



Ilustracja 35: Kod źródłowy (!) programu Piet, który wypisuje frazę "Piet"

czy Sheakespeare, którego kody źródłowe do złudzenia przypominają zapis treści sztuki teatralnej (vide przykładowe "Hello World" z wyjaśnieniem zamieszczone w części załącznikowej pracy - Załącznik 4: "Hello World" (kod źródłowy Sheakespeare)).

Niezależnie od swojego charakteru *języki ezoteryczne* udowadniają, że sposób realizacji tekstu kodu źródłowego jest arbitralny - taka, a nie inna postać klasycznych języków programowania wynika z przyczyn pragmatycznych co powoduje, że mogą i powinny być one przedmiotem szczegółowych studiów.

Przykład: Kod źródłowy jako poezja

Trzecim rodzajem nietypowych praktyk programistycznych, które wskazują na tekstualność medium jest zjawisko tworzenia poezji kodu źródłowego. W tym przypadku

mamy do czynienia z sytuacją o stosunkowo jasnych regułach, które np. strona internetowa Source Code Poetry[50] wylicza:

- Java, C#, CC++ (czyli - *prawdziwy* język programowania);
- musi się kompilować (czyli - musi być *faktycznym* kodem);
- musi się rymować;
- musi być *cool;*

Oczywiście warianty zasad mogą się różnić w zależności od kręgu autorów (wszak np. rymy nie są dziś wyznacznikiem dla bycia poezją). W *call for poets* do tomiku *code{poems}* mającego na celu "eksplorować potencjał kodu do komunikowania na poziomie poezji"[51] wydanego niszowym nakładzie w 2013 roku stawiano tylko dwa wymagania:

- 0,5 KB maksymalnego rozmiaru
- musi się kompilować

Pochodzący z publikacji wiersz Paula Illingwortha *DAILYGRIND* napisany w Javie wygląda następująco[52]:

```
import java.util.Date;

public class DailyGrind {

     public static final void main(String[] args) {

          boolean its_time_to_go_home = false;
          boolean away_the_hours = true;

          while (away_the_hours) {
```

[50] http://sourcecodepoetry.com/ (dostęp 3.08.2014)

[51] http://code-poems.com/project.html (dostęp 3.08.2014)

[52] źródło: http://continentcontinent.cc/index.php/continent/article/download/97/pdf (dostęp 3.08.2014)

```
                Date now = new Date();
                its_time_to_go_home = now.getHours() > 17
                && now.getMinutes() > 30;

                if (its_time_to_go_home) {
                     break;
                }

                try {
                     Thread.sleep(60000);
                } catch (InterruptedException e) {
                     // ignore
                }
           }
      }
}
```

Oczywiście zawsze możemy takiej formie przyjrzeć się z perspektywy programistycznej i stwierdzić, że wiersz ten nie jest *dobrym* kodem bo:

- nie jest obiektowy w zasadniczo obiektowej Javie;
- nazwy zmiennych nie są konsekwentne ;
- przechwycenie wyjątku (*exception)* i zignorowanie go jest karygodne;
- instrukcja `break` użyta do przerwania pętli jest bardzo niechętnie stosowana w Javie (wiąże się z nią dużo problemów bo jest "brutalna", przerywa pracę natychmiast);
- algorytm jest niepoprawny dla soboty, niedzieli i świąt;

Pomimo, że kod jest poprawnym programem napisanym w prawdziwym językiem programowania - funkcja estetyczna (poetyckiego wyrazu) góruje w nim nad pragmatyką tworzenia *softwaru*.

Problem interpretacji

Immanentną właściwością oprogramowania jest fakt bycia przeznaczonym do specyficznego rodzaju interpretacji - połączonej z *wykonaniem jego instrukcji.* Ten *cybernetyczny* aspekt działania softwaru powoduje, że poświęcenie nadmiernej uwagi wyłącznie aspektom tekstualnym funkcjonowania kodu komputerowego byłoby istotnym błędem (powiedzieć - niedopatrzeniem - to stanowczo za mało). Co w takim razie powinno być przedmiotem interpretacji w przypadku refleksji w dziedzinie *studiów nad oprogramowaniem?* W artykule *Critical Code Studies* Mark C. Marino odpowiada:

"Wszystko! Kod, dokumentacja, komentarze, struktury - całość będzie otwarta na interpretacje. (...) W ramach CCS jeśli kod jest częścią programu lub paratekstu (rozumianego szeroko) - współtworzy on znaczenie. (...) W ramach kodu znajdować się będą faktyczne symbole ale również - szerzej - procedury, struktury i gesty. Będą ślady pragamatycznych wyborów dokonywanych przy konstrukcji programu, metody wybrane między innymi i konotacje.

Jako dodatek do symboli i znaków w plikach programu, paratekstualne właściwości będą również istotne dla poinformowanego czytelnika. Historia programu, autor, język programowania, gatunek, źródło finansowania badań i rozwoju (czy to militarne, przemysłowe, rozrywkowe czy inne) - wszystko to wpływa na znaczenia aczkolwiek pojedyncze czytanie może położyć nacisk na tylko niektóre ze wskazanych aspektów."

(Marino, 2006, p. 8)

Naczelną ideą wydaje się więc oddanie sprawiedliwości *softwarowi* jako bytowi złożonemu, funkcjonującemu w pewnym - nie dającym się usunąć - kontekście. Będącemu fenomenem performatywnym, transformatywnym i medialnym zarazem.

Przykład: `10 PRINT CHR$(205.5+RND(1)); : GOTO 10`

Taki tytuł nosi wydana pod koniec 2012 roku i opublikowana za darmo w Internecie książka będąca wspólną pracą (a nie jedynie zbiorem tekstów) czołowych autorów kojarzonych z *software studies* i pokrewnymi dziedzinami: Nicka Montforta, Patsy Baudoin,

Johna Bella, Iana Bogosta, Jeremy'ego Douglassa, Marka C. Marino, Michaela Mateasa, Casey Reas, Marka Sample and Noah Vawtera (vide: Nick Montfort, 2012). Niezwykłość tej publikacji należącej niewątpliwie do dziedziny Critical Code Studies (co zresztą wprost deklarują autorzy) polega na tym, że w całości poświęcona jest tytułowej linii kodu języka programowania BASIC na platformę Commodore 64.

Historia kodu `10 PRINT CHR$(205.5+RND(1)); : GOTO 10` sięga lat 1980-tych kiedy to - jak wiemy - ogromną popularnością cieszyły się komputery osobiste wspomnianej marki. Jednym z podstawowych języków otwierających możliwość samodzielnej pracy użytkowników z tym komputerem był BASIC. Znacznie bardziej przyjazny i wymagający istotnie mniejszej wiedzy techniczno-matematycznej niż asembler stał się niesłychanie popularny wśród posiadaczy C64 (a także Atari i innych platform 8-bitowych). Można bez dużego ryzyka stwierdzić, że większość osób czynnie używających tego komputera znała lepiej lub gorzej BASIC'a. Do popularnych ćwiczeń/zabaw z programowaniem należało tworzenie softwaru generującego różnego rodzaju interesujące wzory, mozaiki i fraktalne figury - tak właśnie działa analizowana przez badaczy linia kodu:

`10` - to oznaczenie numeru linii, w BASICU bardzo często "odliczało" się dziesiątkami tak, aby w razie potrzeby można było dodać kod między wcześniejszymi wersami bez konieczności numerowania od nowa wszystkich kolejnych (nie dało się tego robić automatycznie). Numery wersów były potrzebne ponieważ można się było do nich odwoływać np. instrukcją `GOTO`.

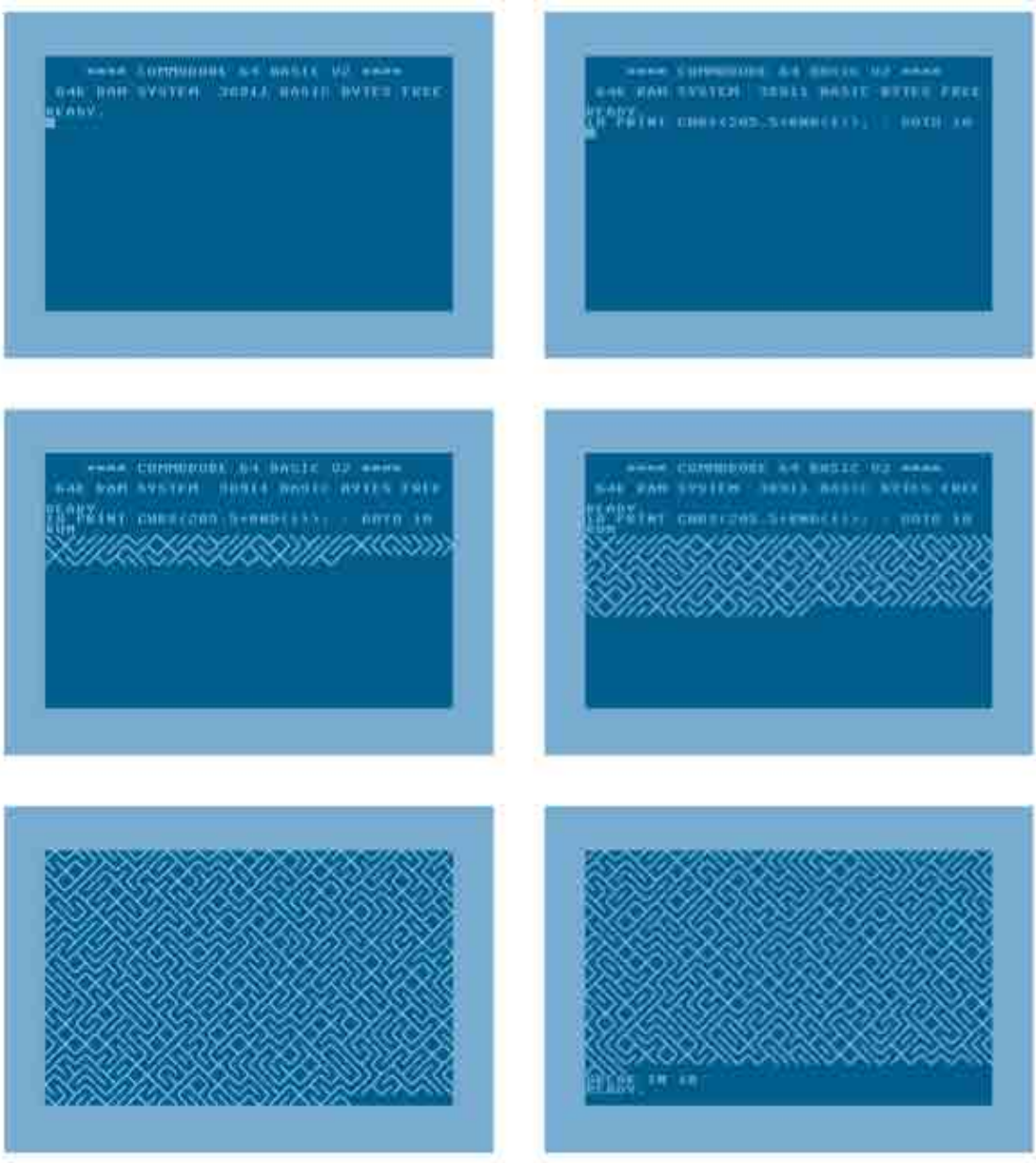
`PRINT` - powoduje wypisanie swojego argumentu na ekranie.

`CHR$` - dokonuje konwersji liczby z przedziału 0-255 na znak ASCII.

`(205.5+RND(1));` - to argument funkcji `CHR$`. Do liczby 205.5 dodawana jest psudolosowa (`RND`) liczba zmiennoprzecinkowa z przedziału 0 do 1. Ponieważ `CHR$` przyjmuje wartość typu w postaci liczby całkowitej - program sam dokonuje zamiany. W rezultacie w wyniku operacji `(205.5+RND(1))` otrzymamy liczbę 205 (dla `RND(1)` < 0.5) lub 206 (dla wylosowanego `RND(1)` > 0.5) co powoduje że otrzymany znak będzie miał postać **** (`CHR(205)`) lub **/** (`CHR(206)`). Średnik kończy pojedyncze wyrażenie - logiczną "linijkę" kodu.

`GOTO 10` - powoduje przejście programu do wersu 10 i wykonanie znajdującej się tam linii kodu, a w efekcie zapętlenie programu.

Skutki uruchomienia powyższego kodu źródłowego poleceniem `RUN` (aż do przerwania jego pracy przez użytkownika) przedstawia ilustracja:



Ilustracja 36: Wprowadzanie kodu języka BASIC dla C64 (1-2),
rezultaty działania programu (3-5),
do momentu przerwania przez użytkownika (6).
(źródło: Nick Montfort, 2012, p. 2)

Na podstawie zaprezentowanego powyżej drobnego programu autorzy publikacji dokonują rozbudowanej analizy sposobu funkcjonowania kodu z uwagi na cały szereg różnych aspektów. Na przykład zaprezentowane zostają jego wariacje, porty na inne platformy, poddana analizie jest kwestia regularności i powtarzalności w różnego rodzaju obiektach informatycznych. Pojawia się propozycja przepisania kodu w Processingu (np. z użyciem możliwości obrazowania 3D dla tego środowiska). Refleksji poddaje się również pojęcie losowości i pseudolosowości w systemach elektronicznych. Osobny rozdział poświęcony jest językowi BASIC - jego historii, znaczeniu dla rozwoju domowej komputeryzacji. Komplementarną wobec *software studies* dziedziną wiedzy są *platform studies* czyli refleksja nad hardwarem - szczególnie istotna z perspektywy historycznej. Stąd nie dziwi obecność w książce tekstu poświęconego komputerom Commodore 64. Ostatecznie *10 PRINT CHR$(205.5+RND(1)); : GOTO 10* jawi się jako tytuł nie tylko o ogromnej wartości poznawczej lecz przede wszystkim jako pionierska, duża publikacja z dziedziny Critical Code Studies. Jednym z istotnych nurtów w szeroko rozumianych *studiach nad oprogramowaniem* są badania potężnych systemów informacyjnych (takich jak Internet) czy analizy gigantycznych zbiorów danych (paradygmat *Big Data* czy projekt *cultural analytics* Lva Manovicha). Autorzy omawianego tomu natomiast - pod pretekstem analizy jednej linii kodu w archaicznym języku programowania dokonują niezwykle interesującego wglądu w naturę i kulturę oprogramowania zgodnie z postulatem, że:

"Jak pamiętniki z zapomnianej przeszłości, komputerowe kody są zanurzone w opowieść o tworzeniu oprogramowania, jego celach, założeniach i więcej. Każdy symbol zawarty w programie może pomóc naświetlić te opowieści i otworzyć historyczne i krytyczne obszary badań. Powszechny odbiór zjawiska oprogramowania może prowadzić do przekonania, że nauka czytania kodu to nudny, matematyczny obowiązek. Mimo to, w wyłaniających się metodologiach krytycznych studiów nad kodem, studiów nad oprogramowaniem i studiów nad platformami - kod komputerowy jest traktowany jako kulturowy kontekst odzwierciedlający historię i społeczne realia swojej kreacji".

(źródło: Nick Montfort, 2012, p. 3)

Αναλυτικά 6: Dane, wyszukiwanie.
Potop informacyjny, Sieć Semantyczna*, Calculus Ratiocinator i Wolfram Alpha.*

"To kuszące, aby traktować silniki wyszukiwania jako samodzielną technologię, wynalazek który ułatwia odszukiwanie rzeczy znajdujących się w innej niezależnej technologii: World Wide Web. Lecz nawet powierzchowna refleksja wykazuje, że silniki wyszukiwania, jak większość innych technologii nie są czymś co może być rozważane w oderwaniu od szerszego społecznego kontekstu, a także od ewolucyjnych zmian społecznych i kulturowych. Wyszukiwarki, z pewnością nie będące odizolowanym współczesnym artefaktem, stanowią probierz cyfrowej kultury i odbicie świata, w którym funkcjonują."

(Halavais, 2009, p. 5)

"Systemy wykorzystujące metody symbolicznej reprezentacji i przetwarzania wiedzy są nieustannie rozwijane. W ostatniej dekadzie jednym z zagadnień znajdujących się w centrum zainteresowań SI były klasyczne problemy optymalizacji wyszukiwania rozwiązań (ang. *search problems*). Z jednej strony rozwijane są metody heurystycznej optymalizacji szukania rozwiązań (ang. *heuristic search*) z drugiej zaś doskonalone są metody optymalizacyjne używane w systemach automatycznego dowodzenia typu SAT (ang. *SATisfiability problems*) i szukania z ograniczeniami, CSP (ang. *Constraint Satisfaction Problems*).

Poszukiwanie właściwych metod reprezentacji wiedzy zapewne nigdy się nie skończy. Reprezentacja wiedzy dostarcza języka, w którym opisujemy problem, który chcemy rozwiązać przy pomocy systemu SI. Dobór języka, w tym jego siły wyrazu, wpływa na to, jakie problemy jesteśmy w stanie skutecznie rozwiązać"

(Nalepa, 2011, p. 392)

Przypomnijmy sobie z rozważań prowadzonych na początku tej pracy pojęcie danych jako bazy dla informacji. Dowiedzieliśmy się wtedy, że stanowią one - ufundowaną na nieredukowalnej, ontologicznej niejednorodności rzeczy podstawę (*rozróżnienie*). Tak rozumiane dane mogą być *formowane* w dające się interpretować struktury - być nośnikami *znaczeń.* Z punktu widzenia metafizyki najważniejszą właściwością danych jest

konieczność ich materialnej implementacji (*bit from it* - dane muszą być w czymś "wyryte"), z punktu widzenia teorii informacji natomiast najbardziej istotne jest spostrzeżenie, że fakt zamieszczenia danych w takim, a nie innym ośrodku ma znaczenie *czysto pragmatyczne.* Oznacza to, że z perspektywy teoretycznej nie jest istotne czy mamy do czynienia z wykutymi w kamieniu kropkami i kreskami czy zmianami stanu magnetycznego na wirującym dysku z pokrytego specjalną substancją stopu aluminium. Rzecz natomiast staje się niesłychanie istotna gdy chcemy uzyskiwać dostęp do ilości informacji liczonych w kilo, mega i gigabajtach na sekundę. Wraz z pojawieniem się nauk informatycznych i całego uniwersum IT *(Information Technologies)* wyznacznikiem zmian jakie zaszły w świecie staje się swoistego rodzaju efekt skali: unifikacja sposobów zapisu i odczytu danych, protokołów ich przetwarzania a także hiperkonektywność urządzeń komunikacyjnych powoduje, że przetwarzalne dane występują w różnych obszarach ludzkiej aktywności w ilościach dotąd niespotykanych. Dodatkowo, coraz większy zakres tego co nazywamy potocznie *realem* jest na różne sposoby włączany w świat przetwarzania informacji. Podstawowym problemem wynikającym z takiego stanu rzeczy jest kwestia "ogarnięcia" - zindeksowania, otagowania, uporządkowania, przechowywania, przetwarzania gigantycznej ilości informacji, które dodatkowo przyrastają w tempie lawinowym.

Powróćmy na chwilę do czasów historycznych, epoki Leibniza, Wilkinsa i Towarzystwa Królewskiego - czyli późnego XVII wieku. Jest to okres kiedy w naszym kręgu cywilizacyjnym nie bez trudności zaczyna kształtować się paradygmat nowoczesnej nauki - empirycznej, racjonalnej, opartej na dowodzeniu i wnioskowaniu według ściśle zadanych reguł. Jest jednocześnie czas pierwszego potopu informacyjnego - naukowcy i filozofowie stają się mobilni w niespotykanym od czasów cywilizacji greckiej, rzymskiej i arabskiej zakresie, kwitnie między nimi korespondencja, z odległych kolonii i nowych kontynentów spływają coraz bardziej frapujące i niezwykłe doniesienia o odkryciach geograficznych, botanicznych, zoologicznych, geologicznych i innych. W Anglii swoje teorie formułuje Isaac Newton (skądinąd wyśmienity przykład "człowieka przełomu", który uznawany jest za ojca nowoczesnej fizyki, podczas gdy jego głównym obszarem zainteresowania była alchemia), mechanizmy buduje Robert Hooke, John Locke formułuje postulaty radykalnego empiryzmu, na kontynencie zaś działa m.in. Gottlieb Leibniz o którym mówi się, że był ostatnim człowiekiem, który orientował się w całości dostępnej w swoich czasach wiedzy (od górnictwa po teologię). Z tej atmosfery intelektualnego

fermentu wyłania się potrzeba zbudowania narzędzia do katalogowania, a przede wszystkim - przetwarzania - gromadzonej wiedzy.

Pierwszą metodą zapewnienia dostępu do repozytoriów wiedzy było tworzenie bibliotek, działalność posiadająca tradycje sięgające co najmniej czasów antycznych. Systematyczne gromadzenie zbiorów to tworzenie swego rodzaju *bazy danych* - skuteczne korzystanie z takiego tworu zakłada nie tylko zebranie informacji w "jednym miejscu" lecz przede wszystkim ich organizację umożliwiającą szybki dostęp i przeszukiwanie zawartości. Nic dziwnego, że system biblioteczny był jedną z licznych fascynacji myślącego jak wiemy bardzo "informatycznie" Leibniza. Opracowany przez niego na potrzeby księgozbioru księcia Hanoweru system katalogowy jest dzisiaj zwany klasycznym (inaczej to: numeryczny system katalogów rzeczowych).

Biblioteka to sposób na zgromadzenie bazy wiedzy i jej dość sprawne indeksowanie. Kluczowa jednak dla skutecznego operowania informacjami, których przyswojenie pochłonęłoby czas większy niż przewidywana długość ludzkiego życia jest możliwość wprowadzenia systematyki między "kwantami" wiedzy i informacji, śledzenia relacji między nimi. Do takich projektów należała *Ars Signorum* George'a Delgarna jak i opisywany już we wcześniejszych rozdziałach język uniwersalny Johna Wilkinsa. Należy w tym miejscu zwrócić uwagę, że były to projekty tyleż lingwistyczne co epistemologiczne - chodziło o ustrukturyzowanie ludzkiej wiedzy i poznania zgodnie ze stanem aktualnie funkcjonującej metafizyki:

"Gatunki Wilkinsa nie powstają wyłącznie w rezultacie łącznia rodzajów i różnic, ale okazują się też nazwami używanymi jako haczyki do zawieszania definicji encyklopedycznych. (...)

Joseph-Marie Degerando w *Des Signes* (1800) zarzucił Wilkinsowi nieustanne mieszanie klasyfikacji z podziałem:

> Podział różni się od klasyfikacji tym, że ta ostatnia budowana jest na podstawie właściwości wewnętrznych przedmiotów, które stara się rozdzielić, tymczasem podziału dokonuje się według pewnych celów, z którymi wiążemy te przedmioty (...) Przykładem jaśniejszym niech będzie gotowa do bitwy armia, każda dywizja pod swoim generałem, każdy pułk pod swoim pułkownikiem (...)

Degerando ma niewątpliwie na myśli koncepcję idealnej biblioteki Leibniza i strukturę *Encyclopédie* (...) to znaczy kryterium podziału dyscyplin według ważności, jaką mają one dla nas."

(Eco, 2013, pp. 277,289-280)

Można by więc używając współczesnego języka powiedzieć, że to co chciał uzyskać Wilkins przypominać miało semantyczną bazę danych wyposażoną w uniwersalny język dostępu, tym zaś co udało mu się stworzyć był specyficzny rodzaj hipertekstu. Nawet bardzo powierzchownie naszkicowana perspektywa historyczna pozwala nam przyjrzeć się temu jak intelektualna elita naszej cywilizacji usiłowała poradzić sobie z pierwszym dużym zalewem informacji przy użyciu dość wyrafinowanego aparatu logiczno-pojęciowego jakim dysponowała lecz wobec braku narzędzi ("hardwaru"), które pozwoliłyby ich przetwarzanie zautomatyzować, a więc wstępnie wielość okiełznać.

Dziś, od kilkudziesięciu lat, jesteśmy świadkami nowego potopu informacyjnego, który jednak jest "kataklizmem" o znacznie większej skali i bardziej doniosłych konsekwencjach - już nie tylko lokalnych, dla pewnych intelektualistów z bardziej uprzywilejowanych rejonów, lecz dla całego świata. Jaki pisze Pierre Lévy:

"Zalew informacji będzie trwał bez końca. Arka nie osiądzie na górze Ararat. Drugi potop nigdy się nie skończy. Ocean informacji jest bez dna. Potop to kondycja, z którą musimy się pogodzić. Nauczmy nasze dzieci pływać, utrzymywać się na powierzchni, a może nawet sterować. Wśród powszechnego falowania jedno jest pewne: Idea stałego lądu jest przedpotopowa"

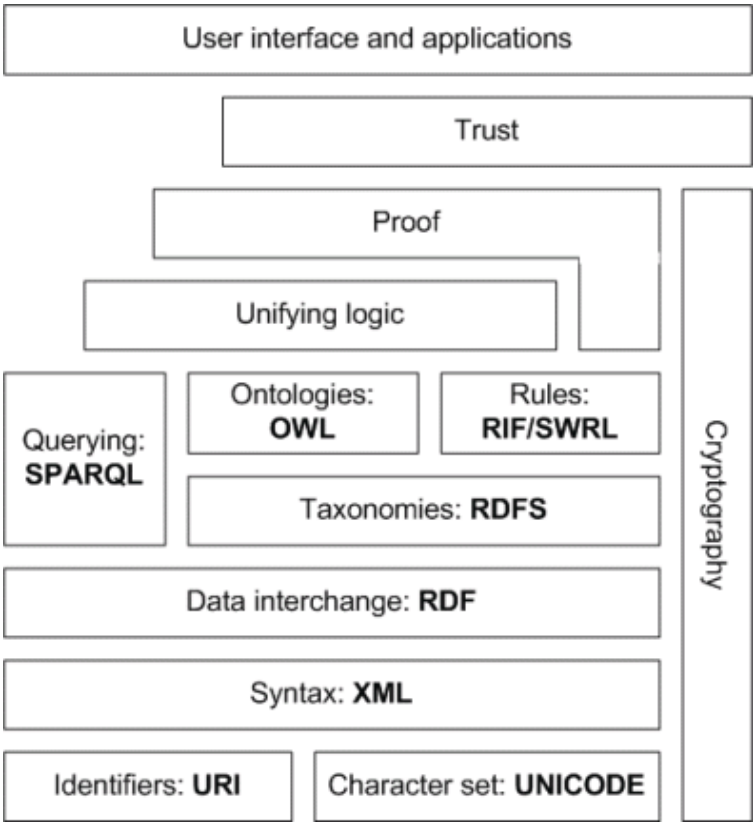
(Levy, 2005, p. 374)

Opisywanym przez francuskiego filozofa "potopem" są oczywiście mass media - w szczególności natomiast Internet. Nade wszystko zaś przenośnia staje się adekwatna po wkroczeniu tegoż w etap rozwoju nazywany niekiedy Web 2.0, którego podstawowym wyznacznikiem jest zatarcie podziału miedzy nadawcą-twórcą a odbiorcą-konsumentem treści i informacji. Kamień milowy dla tego procesu należałoby umieścić zapewne około roku dwutysięcznego, kiedy to wyłonił się szereg narzędzi wspierających tego rodzaju aktywność.

Kontynuując marinistyczne porównanie (skądinąd mające długą tradycję - wszak *cybernetyka* wiedzie swój rodowód od *kybernētēs - sternika* z platońskich dialogów), możemy powiedzieć, że dziś dysponujemy swego rodzaju *flotą*, która pozwala nam dość skutecznie funkcjonować w *zalanym* informacjami krajobrazie. Składa się ona przede wszystkim z mechanizmów i technologii pozwalających użytkownikom na synergiczne edytowanie, tagowanie, redagowanie, udostępnianie różnego rodzaju informacji. Ciężar odpowiedzialności przeniósł się, ze wszystkimi tego konsekwencjami, na sieć społeczności i poziom interakcji międzyludzkiej. Tym samym pierwotny program, marzenie filozofów o obiektywizującym systemie gwarantującym dostęp do wiedzy i stojącym na straży poprawności inferencji nie doczekał się początkowo swojej wielkoskalowej implementacji. Nie oznacza to jednak, że został porzucony. Idea systemu samodzielnie wyciągającego wnioski, porządkującego wiedzę, wyposażonego w obiektywne kryteria rozstrzygania prawdziwości musiała tymczasem ustąpić bardzo prostym w realizacji narzędziom, które pozwalają w umiejętny sposób spożytkować kolektywną działalność milionów użytkowników korzystających z już istniejącej infrastruktury sieciowej - przyjrzymy się tego rodzaju zjawiskom w trzeciej części pracy. W ostatnich latach jednak pojawił się szereg projektów, które podjęły podobną do wilkinsowsko-leibnizowskiej konstrukcji zewnętrznego (na wpół autonomicznego) urządzenia do przetwarzania "pływów" informacyjnego potopu próbę. Należą do nich tzw. **Web 3.0** czyli sieć semantyczna, a także współczesny *calculus ratiocinator* w postaci **silnika Wolfram Alpha**.

Sieć semantyczna to projekt powstający przede wszystkim pod auspicjami W3C (World Wide Web Consortium) z ogromnym zaangażowaniem ze strony jej przewodniczącego, nazywanego czasem *ojcem www* Tima Bernersa-Lee. Chodzi o częściowe przebudowanie dotychczasowej infrastruktury sieci (warstwy softwaru i protokołów, nie sprzętu) tak aby informacje mogły być w jej ramach przetwarzane ze względu na *znaczenie* jakie ze sobą niosą i relacje w jakie wzajemnie wchodzą. Dotychczasowe metodologie budowania spójnej bazy wiedzy na fundamencie zebranych w Internecie treści zakładają raczej podejście *ilościowe*. Tak działa algorytm Google Page Rank, tak funkcjonują również podstawowe narzędzia Web 2.0: wikinomia, folksonomia, społecznosciowe tagowanie i inne. Web 3.0 jest propozycją wprowadzenia standaryzowanych protokołów z metadanymi, dzięki którym odpowiednie *silniki* mogłyby przetwarzać opisywane dane tak jakby je *rozumiały.* Zgodnie z tym programem chodzi więc zarówno o klasyfikację jak i operacjonalizację kolektywnej wiedzy Internetu i to za pomocą systemu, który stwarza pozory obiektywności. Przedsięwzięcie to jest oczywiście

niesłychanie skomplikowane do zaprojektowania, a jeszcze trudniejsze do wprowadzenia, gdyż o ile da się dość łatwo budować zamknięte, spójne systemy umieszczając w nich silną bazę faktów i reguły inferencji (jak w systemach ekspertowych czy semantycznych wiki), o tyle uporządkowanie, certyfikowanie i regulowanie gigantycznego targowiska idei jakim jest globalna Sieć wydaje się wysiłkiem syzyfowym. Żeby jednak podjąć taką próbę (lub skromniej - w celu budowania pojedynczych semantycznych *wysp* w internetowym *potopie* treści) musi zostać opracowany pewien model. Jego podstawowe założenia są zawarte w specyfikacji *Semanic Web Stack* zwanej czasem *Semantic Web Layer Cake:*



Ilustracja 37: Semantic Web Layer Cake[53]

Powyższa ilustracja prezentuje technologie, języki, metody i idee, które umożliwiają działanie semantycznej sieci:

U samych podstaw mamy więc WWW i identyfikowane w niej za pomocą standaryzowanych kluczy (**URI**) zasoby. Takim identyfikatorem może być URL poszerzony o nazwę zasobu (czyli np. http://mojastrona.pl/wiki/Mój_test), specyficznym URI jest też np. ISBN (*International Standard Book Number*).

[53] źródło: http://upload.wikimedia.org/wikipedia/en/3/37/Semantic-web-stack.png (dostęp 5.08.2014)

Drugim elementem jest uniwersalny sposób zapisu liczb i liter. Najpopularniejszym obecnie standardem jest **unicode**, który pozwala na transkrypcję w różnych alfabetach i za pomocą różnych znaków oraz styli.

Jednym z najważniejszych elementów całego układu jest schemat **RDF** (*Resource Description Framework*) pozwalający na XML'owy opis zasobu. Przykładowy zapis wgląda następująco:

```
<?xml version="1.0"?>
<RDF>
<Description about="http://moja_strona_naukowa.pl">
     <author>Jan K. Argasinski</authot>
     <created>january 2014</created>
     <title>Software studies</title>
</Description>
</RDF>
```

Następnie: **RDFS** (*RDF Schema*) dostarcza słownika dla RDF, pozwala na konstruowanie hierarchii klas i właściwości. **SPARQL** jest językiem zapytań dla RDF, w ramach którego można pozyskiwać informacje z baz zgodnych z tym standardem do dalszego przetwarzania. **RIF/SWRL** służy do wymiany reguł (jeśli z jakiegoś powodu dane zestawy wykraczają poza podstawową ontologię).

OWL (*Web Ontology Language*) jest z perspektywy naszych zainteresowań konstrukcją najciekawszą, stanowi bowiem swoistą rodzinę języków służących do powoływania **ontologii**, które rozumiemy tu jako specyfikacje formalne danych zakresów wiedzy. Pozwalają one na uporządkowanie struktur informacyjnych, ułatwiają ich kontekstualizacje i wyrażanie. Najważniejsze zaś, że dzięki ontologiom możemy przeprowadzać wnioskowania, budować modele problemów czy procesów i zasilać je danym z zewnętrznych systemów.

Wyższe warstwy wiążą się z kwestiami gwarantowania poprawności danych i schematów - dostarczaniem dowodów (**Proof**) i zapewnieniem bezpieczeństwa całego systemu (**Cryptography**) co prowadzi do ogólnego ukonstytuowania zaufania do całego układu

(**Trust**). O ile w przypadku zamkniętych systemów budowanych na potrzeby pojedynczych projektów, przedsiębiorstw, mechanizmów wspierających podejmowanie decyzji etc. zaufanie nie stanowi problemu (administratorzy mają ścisłą kontrolę nad danymi, którymi zasilany jest silnik wiedzy) o tyle w przypadku Internetu jako takiego rzecz staje się bardzo kłopotliwa, gdyż jest on niezwykle daleki od spójności w kwestii podejścia do "prawdy", "wiedzy", "faktów" itp. Problem ten pozostaje przedmiotem badań i nie ma jednego uniwersalnego rozwiązania (nie jest [jeszcze] standaryzowany w specyfikacji Semantic Web).

Innym interesującym przykładem próby zorganizowania wiedzy w konstrukt poddający się nie tylko skutecznemu wyszukiwaniu ale również bardziej zaawansowanej komputacji jest silnik *(knowledge engine)* **Wolfram Alpha**, należący do tego samego ekosystemu co opisywany już wcześniej Wolfram Language. Program ten stanowi połączenie potężnej bazy danych, aplikacji webowej (takiej, która działa na serwerach i udostępnia rezultaty przez interfejs www lub innego rodzaju dedykowany *front end*) oraz wyszukiwarki internetowej. Tak skonstruowana usługa usiłuje dokonać *interpretacji* wprowadzonego zapytania, czasem proponując wybór z wielu możliwych zakresów (np. wpisując "orange" zostanie nam podana definicja i przykład koloru, a także zdjęcie z opisem owocu itp.)[54]. Wolfram *potrafi* też dokonywać kalkulacji (wpiszemy: 2+2, odpisze: 4; to samo odnosi się oczywiście do złożonych działań i formuł matematycznych i fizycznych) oraz wykorzystywać wzorce dokumentów, procesów etc. (np. po wpisaniu "mortgage 100000$" silnik zapyta nas o okres pożyczki, oprocentowanie i zwróci prognozowane modele dotyczące spłaty kredytu hipotecznego na bazie aktualnych danych z systemów bankowych). Co więcej można *zasilać* go zewnętrznymi danymi w postaci baz, plików, dokumentów XML itp. Ponadto Wolfram Research udostępnia API (*Application Programming Interface*) i SDK (*Software Development Kit*) czyli środowisko oraz zbiór reguł i specyfikacje, dzięki którym we własnych programach czy stronach internetowych developerzy mogą stosować ten uniwersalny silnik komputacji wiedzy. Czyżby więc spełnił się Leibniza sen o *lingua characteristica* i c*alculus ratiocinator,* czyli uniwersalnym języku i maszynie, które pozwoliłyby na "mechaniczne" rozstrzyganie sporów? Volker Peckhaus streszcza rozważany aspekt koncepcji niemieckiego filozofa w następujących zagadnieniach:

---

[54] przykłady za: http://www.wolframalpha.com/tour (dostęp: 5.08.2014)

- *characteristica universalis* zakłada, że posiadamy kompletną listę "prostych myśli", elementów podstawowych, "kwantów wiedzy";
- "proste myśli" mają być jednoznacznie przypisane znakom (im mniejsza lista tym łatwiej to zrobić);
- kompletna realizacja *characteristica universalis* może być niewykonalna lecz nie przeszkadza to aby służyła ona za regulatywną ideę - rzeczywiste systemy symboliczne mogą być stosowane do skutecznego rozwiązywania przynajmniej części problemów;
- powyższe nie jest rezultatem ograniczenia co do zasady lecz jedynie ograniczeń ludzkich możliwości w tym zakresie;
- *calculus ratiocinator* służy do mechanicznego dedukowania wszystkich możliwych prawd z listy "prostych myśli";
- formuje on syntaktyczną część *lingua rationalis*, zaś *characteristica* odpowiada za część semantyczną*;*
- wstępny okres częściowych realizacji programu systemu znaków dopuszcza wielość interpretacji - w finalnym (utopijnym) etapie, po osiągnięciu pełnej listy desygnatów wszystkich możliwych "prostych myśli" system staje się pewny i kategoryczny;

(za: Peckhaus, 2005, pp. 181-182)

W świetle tak przedstawionej koncepcji po raz kolejny staje się oczywiste, że nowożytna racjonalno-empirycza filozofia europejska dysponowała już bardzo silnymi intuicjami na temat tego jak radzić sobie można z przetwarzaniem wiedzy i informacji. Zabrakło oczywiście dwóch elementów: uniwersalnego systemu zapisu danych (cyfryzacja) oraz odpowiednio wydajnego hardwaru (warto tu wspomnieć kłopoty lorda Babbage z *maszyną różnicową,* której zbudowanie - pomimo gotowych planów i działających modeli okazało się zbyt skomplikowane inżynieryjnie... i finansowo). Tym, czego nie przewidzieli dawniejsi myśliciele jest skala i dynamika z jaką rozwija się infosfera. W naszym wypadku ilość jednakowoż zdecydowanie przechodzi w jakość i ma ogromne znaczenie dla całego analizowanego uniwersum. Idee komputera, oprogramowania, sieciowości zyskują zupełnie inny wymiar gdy staną się faktycznymi przedmiotami codziennej manipulacji setek milionów użytkowników wyposażonych w miliardy urządzeń komunikacyjnych. Wybranym implikacjom tych kwestii przyjrzymy się w trzeciej części pracy.

Αναλυτικά 7: Pamięć operacyjna. Pamięć kulturowa.

"A więc czasowość kodu jest znacznie szybsza niż czasowości dnia codziennego. Synchronizowanie komputerów do naszych ekstremalnie powolnych żyć jest często wyzwaniem samym w sobie, ponieważ komputery nie są zbyt dobre w czekaniu. To dlatego interfejs między użytkownikiem a maszyną staje się kluczowy - jest tłumaczącym pośrednikiem pomiędzy pracą kodu a codziennym życiem operatora"

(Berry, 2011b, p. 98)

"Pamięć elektroniczna to każda maszyna lub układ, który utrwala efemeryczne sygnały procesora centralnego. Pierwszy układ pamięci komputera von Neumanna, który zawiera komórki pamięci bezpośrednio dostępne dla procesora jest czasami nazywany »pamięcią wewnętrzną«. W niej to programiści tworzą geometryczne struktury danych. Pamięć wewnętrzna jest nieciągła i składa się z szeregu miejsc magazynowych, z których każde przechowuje jedną jednostkę danych (jeden lub kilka bajtów). Każde miejsce ma swój własny adres, może więc być indywidualnie wykorzystane przez procesor przy odczytywaniu starych danych oraz wpisywaniu nowych. Pamięć wewnętrzna pozwala na dostęp bezpośredni (*random access,* dostęp przypadkowy): procesor centralny może bezpośrednio i szybko przejść do miejsca 1000, bez badania miejsc od 1 do 999"

(Bolter, 1990, p. 229)

Jak zwraca uwagę w innym niż cytowany powyżej fragmencie *Człowieka Turinga* J. David Bolter - pamięć komputerowa, wpisuje się w wiedzioną od starożytności tradycję mnemoniczną i stanowi zarazem w pewnym sensie szczyt jej rozwoju. Cyfrowa reprodukcja danych jest pozbawiona wad kopii analogowych. Zapewniając odpowiednią redundancję cyfrowych nośników informacji jesteśmy w stanie zagwarantować danym praktyczną nieśmiertelność w niezmienionej formie. Jednocześnie w świetle nakreślonych w poprzednich rozdziałach technik możemy stwierdzić, że jako użytkownicy mamy zapewniony, dzięki niesłychanie skomplikowanym urządzeniom ubranym w przyjazne interfejsy, w miarę nieograniczony dostęp do globalnej, usieciowionej pamięci ludzkości. Wendy Chun nazywa taki stan, bardzo obrazowo, "pamięcią *nieumarłą* (undead)" (Chun, 2011, p. 133) - rzecz w tym, że komputerowa pamięć pozbawiona jest (pomimo nazwania tego elementu hardwaru bardzo potężnym i bogatym w konotacje słowem jakim jest właśnie "pamięć") istotnego zdaniem autorki, organicznego, repetycyjnego charakteru.

Warren Sack w haśle poświęconym pamięci w *Software Studies. The Lexicon.* zwraca uwagę na historię, konstrukcje i założenia tego obiektu (Sack, 2008, p. 183). Chodzi o mało znany współcześnie fakt, że słowo "komputer" odnosiło się pierwotnie do pewnego rodzaju zawodu polegającego wykonywaniu obliczeń. Osoby zwane komputerami (w języku polskim najbliższe słowo to prawdopodobnie "rachmistrz") były bezpośrednią inspiracją dla Turinga, jego poprzedników i następców przy konstruowaniu maszyn obliczeniowych. W czasie II Wojny Światowej tzw. Bomby Turinga, protokomputery służące do łamania niemieckich szyfrów stworzone zostały nie po to by wykonać nowy rodzaj zadań lecz aby przyśpieszyć i zwiększyć efektywność pracy zespołów ludzkich *komputerów.* Skądinąd warto podkreślić fakt, że komputerkami były przede wszystkim kobiety - co spowodowało, że po uruchomieniu projektów w rodzaju centrum kryptograficznego Bletchley Park czy późniejszych cywilnych zastosowaniach urządzeń liczących stały się one w sposób naturalny pierwszymi programistkami. Mamy więc do czynienia z sytuacją gdy jak pisze Sack "metafora staje się modelem naukowym" (ibid, p. 134), ma to swoje donioślejsze niż tylko anegdotyczno-historyczne konsekwencje:

"Współcześni teoretycy porównują pamięć ludzką z komputerową. Kognitywiści, którzy eksplorują tą analogię wierzą, że ludzie i maszyny są dwoma gatunkami tego samego rodzaju; słowami informatyka i ekonomisty Herberta Simona ludzie i komputery są »systemami symbolicznymi«. W związku z tym niektórzy kognitywiści hipotetyzują, że ludzka pamięć jest nie tylko podobna do komputerowej lecz że jest generalnie tym samym. Lub, by ująć to inaczej, hipoteza głosi, że pamięć komputerowa nie jest tylko jednym z możliwych modeli ludzkiej pamięci lecz, że jest najlepszym." (ibid.)

Jednym ze śledzonych przez nas wcześniej tropów XX-wiecznej nauki, była kwestia konceptualizacji i implementacji systemów przetwarzania znaków - wiemy, że istnieją dobrze działające generatory ciągów znaków (gramatyki generatywne), które stanowią model komputacyjny. Maszyna Turinga też jest urządzeniem do przetwarzania znaków zgodnie z instrukcjami. Wiemy - w końcu - że pewnego rodzaju wyrafinowaną manipulację bardziej złożonymi systemami symbolicznymi możemy traktować jako rodzaj "rozumienia" czy przejaw "inteligencji", zaś z całą pewnością "rozumowania". Turing tworząc swoją Maszynę (która, przypomnijmy, była konceptem teoretycznym) myślał o sposobie funkcjonowania znanych mu zjawisk, w tym ludzkich *komputerów,* którzy mogli stosować różnego rodzaju mnemoniczne techniki (robienie notatek, abakus, liczmany etc.). "Taśma"

i "głowica" Uniwersalnej Maszyny mają swoją moc wyjaśniającą również dzięki temu, że są odniesieniami do obiektów życia codziennego. Pierwsze komputery (jak zwraca uwagę przytaczany tekst Sacka) musiały być każdorazowo konfigurowane pod rozwiązanie jakiegoś problemu. W tym sensie nie różniły się od pracowników-ludzi, których praca musiała być rozplanowana przez "programistę", rozdysponowana, zaś uzyskane wyniki złożone w spójną całość. Sytuacja zmieniła się dopiero po pojawieniu się opisywanej już wcześniej architektury von Neumanna - odtąd komputery mogły użyć *pamięci* w celu "odłożenia" jakiegoś stanu do późniejszego przetwarzania, zachowania samego programu, który może być użyty w innej części zadania etc. Ferromagnetyczne dyski, na których można było zapisywać stany w postaci różnic pola doprowadziły do paradoksalnej sytuacji w której komputery-urządzenia zyskały większą autonomię w rozwiązywaniu zadań niż mieli komputerzy-ludzie nie zdający sobie do końca sprawy z tego czemu konkretnie służą wykonywane przez nich obliczenia.

Odczytując stan z dysku/taśmy maszyna staje się, jak pisał sam Turing w *On computable numbers* na swój sposób "cyfrowo świadoma" odczytanego znaku. W tym sensie *pamięć komputerowa* jest nie tylko obiektem *magazynującym dane*. Wendy Chun w *Programmed Visions* pisze:

"*Pamięć (memory)* i *kolekcja danych (storage)* to nie to samo. *Memory* wywodzi się od sanskryckiego korzenia *martyr* i wiąże się ze starożytnym, greckim terminem oznaczającym *zgubny, wybredny*. Pamięć zawiera się w akcie powtarzania: jest czynnością upamiętnienia - procesem wspominania lub przypominania sobie. W przeciwieństwie do słowa *store*, które według OED (*Oxford English Dictionary* - dop J.K.A.) wywodzi się od starofrancuskiego *estorer* znaczącego »zbudować, ustanowić, wznieść«" (Chun, 2011, p. 134)

Przytoczone rozróżnienie nie jest wyłącznie językowe. Rzeczywiście istnieje silne rozróżnienie między *pamięcią operacyjną,* a *pamięcią trwałą.* W przypadku techniki komputerowej odczytanie czegoś oznacza tak naprawdę - zapisanie tego gdzieś indziej. Z ośrodka trwałego *(non-volatile)* w bardziej ulotny, operacyjny. Osobnym zagadnieniem, któremu w pewnej części poświęcony jest rozdział Δ2 tej pracy jest kwestia istnienia pamięci usieciowionej, rozproszonej w serwerowej *chmurze* i utrzymywanej dzięki złożonej infrastrukturze Internetowych gigantów.

Tematyce cyfrowej pamięci, ewolucji technik mnemonicznych, a także tego w jaki sposób nasza tożsamość wchodzi w relacje z regulowanymi softwarowo dygitalnymi artefaktami takimi jak portale społecznościowe, blogi i inne miejsca w Sieci, które stanowią swoisty zapis naszej i o nas pamięci był poświęcony festiwal Ars Electronica 2013: Total Recall. Pośród szeregu interesujących projektów, instalacji i performensów poddawanych analizie w czasie towarzyszącej imprezie konferencji pojawiła się również refleksja nad fenomenem o charakterze stricte softwarowym: *Museum of Me*, stworzonym przez firmę Intel.

PRZYKŁAD: Intel Museum of Me

Ilustracja 38: Strona powitalna projektu w widocznym wirtualnym muzeum.

Projekt ten został uruchomiony w 2011 pod adresem internetowym http://www.intel.com/museumofme. Działanie zamieszczonego tam programu polega na łączeniu się za zgodą użytkownika z jego profilem na portalu społecznościowym Facebook i stworzeniu "indywidualnej" wystawy prezentującej osoby, zdjęcia, lokacje i wydarzenia tam opisane. Całość ubrana jest w animowany, pół-interaktywny interfejs stworzony we Flashu. Użytkownik widzi w jednej z sal zawieszone portrety swoich znajomych, w innej zaś oprawione w ramy i zakomponowane zdjęcia ze swojego archiwum. W kolejnej sali widać mapy z miejscami, w których był z dołączonymi zdjęciami i profilami osób itd. itd. Całość skomponowana jest w narrację współczesnego muzeum. Po salach przechadzają się zwiedzający, część z nich "ogląda" multimedialną instalację prezentującą analizę frekwencyjną słów używanych na profilowym *wallu.* Na końcu istnieje możliwość wygenerowania (i wydrukowania) katalogu "wystawy".

Ilustracja 39: "Wejście" na wystawę; Prezentacja znajomych; Część pt. "Moje słowa".

Pod względem technicznym projekt nie należy do najbardziej skomplikowanych. Facebook udziela twórcom oprogramowania rozbudowane API, które może być zastosowane zarówno do tworzenia wewnętrznych aplikacji (takich jak gry, sklepy internetowe, konkursy dostępne po zalogowaniu się do portalu) jak i zasilania zewnętrznego oprogramowania danymi pochodzącymi z serwisu (najczęściej są to rozwiązania uwierzytelniające typu "zaloguj kontem facebook", ale mogą być też użyte dowolne inne informacje zebrane w serwisie - lista znajomych, zdjęcia itp.; oczywiście po uprzedniej autoryzacji). Mając więc bardzo łatwy dostęp do dobrze posegregowanych danych zgromadzonych przez użytkownika na portalu wystarczyło opracować odpowiednią aplikację robiącą z tych danych użytek - tu umieszczając je w odpowiednim interfejsie wizualnym.

Prawdziwą siłą projektu *Museum of Me* jest bardzo artystyczny w swoim wydźwięku akt problematyzacji naszych codziennych aktywności. Teoretycznie każdy, kto korzysta z usług portali w rodzaju Facebooka powinien mieć świadomość faktu bycia obserwowanym. Jednak konstrukcja interfejsu, sposób wprowadzania informacji i reagowania na nie daje raczej poczucie prowadzenia komunikacji o charakterze dialogu, osoba komentująca wpis raczej jest traktowana jako rozmówca niż widz (oczywiście inaczej jest w przypadku profili firm, marek czy fanpage'ów osób publicznych). Postawienie naszych prywatnych, czasem nawet dość intymnych treści w kontekście muzealnym, kiedy widzimy "osoby" stojące w zaaranżowanej przestrzeni, przyglądające się naszym zdjęciom i faktom z naszego życia bardzo interesująco zmienia kontekst naszej aktywności w Sieci. Warto zwrócić uwagę, że w wielu branżowych mediach opisywano doświadczenie *Museum of Me* jako bardzo ciekawe ale i niepokojące *(creepy) zarazem.*[55]

[55] np. PC World (http://www.pcworld.com/article/229138/Intel_Museum_of_Me_is_Cool_Creepy_Facebook_Fun.html [dostęp 5.08.2014]); Mashable (http://mashable.com/2011/06/01/intels-museum-of-me-app/ [dostęp 5.08.2014]); CNet (http://www.cnet.com/news/intels-facebook-museum-of-me-is-a-must-try/ [dostęp 5.08.2014]).

Z przeprowadzonej dotąd refleksji wyłania się kilka aspektów w ramach których możemy analizować zjawisko *pamięci* w odniesieniu do technologii informatycznych. Każdy z nich przekłada się na inny poziom pracy oprogramowania.

Mamy więc pamięć operacyjną i trwałą hardwaru, która przekłada się na taki a nie inny sposób wykonywania kodu w komputerze. Następuje tu niskopoziomowe przełożenie architektury sprzętowej na techniki budowania softwaru. W tym zakresie sensów mieści się też pamięć jako miejsce przechowywania wiedzy o sposobach realizowania algorytmów - programów. Dany software *zajmuje* jakieś miejsce na dysku. Ta warstwa wydaje się być w przeważającej mierze niewidzialna dla końcowego użytkownika.

Pamięć odnosi się również do procesów przetwarzania i magazynowania danych na wyższym, rozpoznawalnym przez użytkownika poziomie. Chodzi o kolekcje danych obecne w pamięci każdego urządzenia komputerowego. Dokumenty tekstowe, pliki graficzne, muzyka w mp3, które oglądane są na zmetaforyzowanych *biurkach i pulpitach*, przechowywane w *katalogach* i *folderach.* Dochodzi tu do konceptualnej transformacji: nośniki danych *pamiętają* je, użytkownicy *korzystają* z nich na kształt przedmiotów. Z podobną sytuacją mamy do czynienia w opisywanym już przypadku obiektowego paradygmatu w oprogramowaniu gdzie *procesualny* z ducha kod źródłowy jest prezentowany jak *rzecz* - obiekt właśnie.

Kolejna z poruszanych kwestii to pamięć uzewnętrzniona w Sieć. Wplątana w szereg ponadjednostkowych i ponadosobowych relacji. *Wrzucając* dane w *chmurę* mamy do nich nadal dostęp (*pamiętamy* je) jednak jednocześnie włączamy je w nierekonstruowalny świat serwerów, klastrów, sieci wymiany informacji. Jeśli użyć metafory Adriana Mackenzie - *karmimy softwarowego Lewiatana* (vide: Mackenzie, 2003a). Prawdziwym problemem, z którym dostawcy usług i użytkownicy Sieci dopiero zaczynają się poważnie mierzyć jest kwestia tego co *pamięta* Internet. Co pozostaje w globalnej sieci kiedy już nas nie ma? Czy osoba, żywa lub martwa ma prawo do bycia *zapomnianym* w Sieci? O żywym charakterze tak postawionych problemów świadczy np. uruchomienie przez Google usługi Inactive Account Manager, która pozwala na przekazanie np. spadkobiercom dostępów i danych naszych nieaktywnych już kont i usług. Warto tu wspomnieć o niedawnych próbach prawnego uporządkowania *prawa do bycia zapomnianym* i zasad na jakich informacje o osobach miałyby "znikać" z wyników wyszukiwania.

Αναλυτικά 8: Przyszłość teraz.
Ubiquitous Computing & Internet of Things.

"Używam pojęcia *rzeczywistość* stosując je do każdej rzeczywistości człowieka. Zjawiska zachodzące w rzeczywistości elektronicznej i rzeczywistości świata fizycznego opisuję jako istniejące, prawdziwe/nieprawdziwe, traktuję te dwa rodzaje bytu podobnie pod względem ich statusu istnienia, czyli jako rodzaju rzeczywistości, wskazując na ich odmienność."

(Myoo, 2013, p. 24)

„»Dzianie się w świecie« to coś, co musi być zrozumiane. Od czasu do czasu i zawsze na czas, nowe formy wyłaniają się i są katalizatorami przemian dotychczasowo istniejących aktorów, rzeczy, fenomenów w czasie i przestrzeni w nowe tryby egzystencji, nowe zestawienia, takie które powodują, że rzeczy działają na inne sposoby produkując i instancjując nowe możliwości. Nowe formy i aktywności czynią – w mniej lub bardziej zaskakujący sposób – pewne rzeczy wyobrażalnymi”

(Rabinow, 1999, p. 180)

"Komputacja jako coś »scentralizowanego, sekwencyjnego i zorientowanego na rezultaty« oraz zasadniczo skoncentrowanego na kalkulacji przemieszcza się stopniowo lecz nieubłaganie w stronę ujęcia »komputacji jako interakcji«."

(S. M. Matthew Fuller, 2011, p. 147)

W świetle naszych dotychczasowych rozważań technologie komputerowe jawią się jako coś bardzo *osobnego.* Zbudowane na bazie wyrafinowanego modelu konceptualnego skomplikowane urządzenia, które służą do przetwarzania dyskretnych danych ujętych w abstrakcyjne struktury, zapisanych jako stany magnetyczne w hermetycznie zamkniętych pudełkach dysków i procesorów. Taki obraz wydaje się czymś bardzo odległym od naszej codziennej praktyki działania w świecie. Tymczasem - wiemy doskonale - komputery otaczają nas we współczesnym życiu zewsząd i w bardzo różnych zakresach. Należą do podstawowych narzędzi komunikowania się, wymiany informacji, pracy, rozrywki etc. Poniższy rozdział poświęcony będzie technologiom obliczeniowym *obecnym* choć często

niewidzialnym w codzienności - bardzo dobrą i opisową metaforą jest tu *zanurzenie* cywilizacji w cyfrowych *odmętach.*[56]

Podstawowe paradygmaty funkcjonowania powszechnie dostępnych technologii obliczeniowych działają w trzech dających się wyróżnić poziomach (które również następowały po sobie chronologicznie):

- *szeroko dostępnych komputerów osobistych*:
  urządzenia komputerowe są tu osobnymi *(stand alone)* stacjami roboczymi do których oprogramowanie *wgrywa* się za pomocą dyskietek, dysków CD, DVD i innych nośników;

- *powszechnego usieciowienia*:
  komputery podłączone w wewnętrzne pajęczyny komunikacyjne *(intranety)* zyskują nowe funkcjonalności oraz po uzyskaniu dostępu do Globalnej Sieci stają się *otwartymi* narzędziami komunikacji;

- *miniaturyzacji i mobilności osobistych, podłączonych do Sieci urządzeń komputerowych*:
  komputery przyjmują postać najpierw zasilanych bateriami PDA czy palmtopów, notebooków a następnie smartfonów, tabletów, laptopów; wyposażone w bezprzewodowe środki komunikacji z Siecią oraz peryferia umożliwiające działanie kontekście (kamery, akcelerometry, systemy pozycjonujące, interfejsy RFID *[rafio-frequency identification],* złącza bluetooth).

Ogromnej skali proces jaki stanowiło przejście ze stanu opisanego w podpunkcie drugim do kolejnego odbył się w bezprecedensowym tempie kilkunastu lat. Co jeszcze bardziej interesujące przebiegł on w sposób bardzo płynny - dzięki zastosowaniu intuicyjnych interfejsów, rozwojowi wiedzy w dziedzinie HCI *(Human Computer Interactions),* a także potężnej machinie marketingowej niesłychanie złożone technologie są adaptowane przez użytkowników bardzo łatwo, naturalnie włączając się w praktyki dnia codziennego. Skutkiem ubocznym takiego stanu rzeczy jest rosnąca *przeźroczystość* mechanizmów rządzących nowymi mediami. Przeciętny użytkownik coraz słabiej rozumie

---

[56] Na marginesie: znów narzuca się nośność *wodnych* metafor w odniesieniu do danych. Mieliśmy cyfrowy *potop* lub *zalew* informacji. Możemy się w ich świecie *zanurzyć* lub podążać za *strumieniami* danych. Współcześnie często korzystamy z *chmury* (ulotna forma wody), zaś cybernetyka to nauka o *przepływie* informacji w procesach *sterowania* (od "sternika"). Dziś już raczej nie mówi się o *surfowaniu* w Internecie, natomiast dość często o *nawigowaniu* (łac. *navigo*, *navigare* - żeglować) w nim.

zasady działania sprzętu, którego nieustannie używa oraz reguły według których pracuje oprogramowanie - widać to wyśmienicie w przypadku dużych zmian wprowadzanych *pod powierzchnią* portali internetowych i silników takich jak Google Search czy Facebook. Pierwszy z nich przeprowadza regularne przebudowy, które zmieniają sposób indeksowania i udostępniania informacji użytkownikom podczas gdy podstawowy interfejs pozostaje praktycznie bez większych zmian od początku działania usługi - składa się z charakterystycznego logo i paska do wprowadzania zapytań pod nim. W drugim przypadku - Facebooka zmiany są znacznie lepiej widoczne, ponieważ pociągają za sobą często ewolucję sposobu prezentowania danych - warto jednak zwrócić na fakt, że np. duża zmiana wprowadzona m.in. wraz z uruchomieniem semantycznego Facebook Graph Search (lato 2013) była powszechnie krytykowana przez jakiś czas (około miesiąca lub dwóch) po czym użytkownicy przeszli nad nią do porządku dziennego. Dziś mało kto potrafiłby wskazać jak wyglądał interfejs największego portalu społecznościowego rok czy dwa lata temu i czym różnił się od aktualnego. Ta użytkownicza adaptacyjność i skoncentrowanie na teraźniejszości powoduje, że technologie perwazywne mogą zacząć działać w świecie wokół nas, zmieniać jego informacyjny charakter i pozostać nie tyle niezauważone (wiemy, że używamy takich a nie innych narzędzi) co *nieuświadomione* w sposób głębszy czy krytyczny.

Ubiquitous computing czy *pervasive computing* tłumaczone na język polski jako *przetwarzanie rozpowszechnione* to stosunkowo nowa dziedzina technologii, które zaprojektowane są w ten sposób by zaangażować urządzenia komputerowe w możliwie szeroki zakres życia codziennego użytkowników. Z *ubicompu* nie korzystamy więc siedząc przy komputerze czy laptopie i dostosowując swoje działania do oprogramowania w celu realizacji pewnych zadań lecz np. poruszając się po mieście z odpowiednim urządzeniem przy sobie (np. telefonem) i aktywując w ten sposób sieci niewidzialnych sensorów, będąc geolokalizowanymi itp. Nie istnieje *jedno* przetwarzanie rozpowszechnione*,* paradygmat ten może obejmować swoim zakresem niektóre z wykorzystań standardowych komputerów i tabletów, Internetu, oprogramowania pośredniczącego (chumura, middlewary), ambientowych interfejsów (Google Glass, smartbandy, "inteligentne zegarki" i inne urządzenia, które nosimy jako części garderoby *[wearable computing]),* a także odpowiednio przygotowanych urządzeń codziennego użytku (pralek, lodówek, samochodów).

"Wiele form *ubiquitous computing* jest nierozróżnialnych z punktu widzenia użytkownika i będzie mu się prezentowało w postaci pojedynczego paradygmatu: *everyware"*

(Greenfield, 2006, p. 15)

Kluczową kwestią dla tej idei jest - jak zwraca uwagę autor w innej części cytowanej powyżej książki - włączenie obiektów, powierzchni i obszarów codziennego użytku w uniwersum przetwarzania informacji. Bardzo istotnym, obok wrażliwości na kontekst *(context awareness)* jest tu również oparcie funkcjonowania aplikacji i hardwaru na ludzkim działaniu, lecz nie w rozumieniu dotychczasowym - kiedy to użytkownik był *operatorem* urządzenia komputerowego lecz w sensie *rozpuszczenia (dissolve)* interakcji człowiek-maszyna w sieci czynności i tak wykonywanych (poruszanie się z miejsca na miejsce, uprawianie sportu, robienie zakupów, spotykanie się ze znajomymi, prowadzenie samochodu itp.). Powoduje to, że interfejsem pracy z oprogramowaniem może stać się (bardzo często nieświadomie) potencjalnie *cała* rzeczywistość w której funkcjonujemy. Działanie ubicompu jest więc skalowalne: może on operować na poziomie pojedynczego człowieka i jego gestów; pojazdu, którym się porusza; budynku, w którym się znajduje; miasta, w którym żyje. *Powszechność* takiego przetwarzania powoduje, że w części przypadków w wątpliwość można poddawać nawet zasadność kategorii "użytkownika" w odniesieniu do tak rozumianej technologii. W publikacji *Ubiquitous computing. Smart Devices, Environments and Interactions* Stefan Poslad wymienia m.in. trzy klasy technologii *aktywujących* przetwarzanie rozpowszechnione właśnie w kontekście skali:

- *Pył (dust):* urządzenia wahające się gabarytami od kilku mikro do milimetrów[57];
- Skóra *(skin):* urządzenia działające na bazie cienkich, elastycznych polimerów i tkanin, które same mogą przewodzić światło (a więc przekazywać informacje) lub prąd elektryczny albo pozwalają na wbudowanie w swoją strukturę elementów przewodzących;
- Glina *(clay):* różnego rodzaju elastyczne, modyfikowalne, wszczepialne i rozproszone urządzenia mogą być formowane w bardziej złożone układy tworząc obiekty medialne charakteryzujące się niestandardowymi sposobami operowania na nich *(tangible interfaces)*;

(za: Poslad, 2009, p. 28)

[57] vide: http://www.cse.wustl.edu/~lu/cse521s/Papers/SmartDust.pdf (dostęp 10.08.2014)

Ubiquitous computing to sposób myślenia i tworzenia oprogramowania w paradygmacie interkonektywnym - rzecz w tym, żeby urządzenia komputerowe *ukryć, zaangażować* w sieć codziennych działań osób, które ich używają a także *połączyć* w sieć w sensie komunikacyjnym. Równie interesującą ideą, która wyłania się z takiego spojrzenia na technologię jest *Internet rzeczy (Internet of things)*, który nieco inaczej kładzie akcenty jeśli chodzi o podejście do szerokiego i usieciowionego przetwarzania informacji. IoT opiera się nie tyle na miniaturyzacji dedykowanych urządzeń komputerowych (choć oczywiście musi ją zakładać) lecz na włączaniu (aktywowaniu) przedmiotów codziennego użytku w infosferę. Różnica jest bardzo subtelna i często nie daje się jednoznacznie stwierdzić. Możemy jednak przyjąć wspomniane wyżej kryterium:

Ubicomp to nowe urządzenia komputerowe: mobilne, ubieralne, wbudowane w obiekty dotychczas nie będące skomputeryzowanymi, *niewidzialne* etc*.*

natomiast

IoT tworzą rzeczy w ten czy inny sposób *włączone* w Sieć za pomocą sensorów, lub interfejsów komunikacyjnych (bluetooth, RFID) same niekoniecznie będące komputerami (np. książki z wbudowanym chipem, który pozwala na liczenie ile razy były zdejmowane z półki w bilbiotece; czujniki wbudowane w uliczne estakady, które pozwalają na śledzenie natężenia ruchu; moduły udostępniające w Sieci informacje o swoim stanie pozwalające np. sprawdzić w telefonie czy drzwi od mieszkania są zamknięte etc.).

Dotychczas, opisując technologie *perwazywne* koncentrowaliśmy się głównie na hardwarze. Miniaturyzacja i mobilność to najbardziej narzucający się aspekt rozwoju IT ostatnich lat. Musimy jednak pamiętać, że wraz ze zmianami w sprzęcie idzie przekształcenie paradygmatów w myśleniu o oprogramowaniu - co z perspektywy tej pracy wydaje się znacznie bardziej istotne. Na potrzeby ubicompu i IoT zostały opracowane specjalne architektury, które koncentrują się na przetwarzaniu elementów dotąd mniej istotnych lub wręcz całkowicie pomijalnych takich jak np.:

- zarządzanie zadaniami na poziomie aktywności użytkownika w kontekście *(task management layer)* gdzie chodzi o indeksowanie czynności podmiotu w środowisku i *odfiltrowanie* spośród nich tych, które są *znaczące* dla systemu *(need of service);*

- zarządzanie warstwą środowiska *(environment management layer)* w celu monitorowania dostępnych zasobów i statystyki stanów użytkownika;

Są one realizowane przez wyspecjalizowane protokoły (takie jak 6LoWPAN, MQTT, CoAP), platformy databrokreskie (Thingworx, Sense) czy *osadzone (embeded)* systemy operacyjne (np. TinyOS, Mantis, LiteOS czy Contiki). Przetwarzanie rozpowszechnione w znaczniej mierze opiera się też na rozwiązaniach z dziedziny Open Source, możemy tu wymienić takie projekty jak:

- *RIOT*: system operacyjny dla Internet of Things;
- *Thingspeak*: API do tworzenia aplikacji IoT przez HTTP *(Hypertext Transfer Protocol);*
- *IoT toolkit*: zestaw narzędzi do budowania komunikacji między protokołami dla IoT;
- *The Thing System*: kolejny open source'owy system z własnym API do IoT;
- *Nitrogen*: platforma dla budowania interkonektywności miedzy urządzeniami;

oraz wiele innych.

Z naszej perspektywy interesujący jest fakt, że ubicomp i IoT pozwala na tworzenie rozbudowanych ekosystemów, które ułatwiają implementację lub też są bezpośrednio gotowe do użycia. Poniższe przykłady prezentują dwa godne uwagi projekty rodem z Polski.

PRZYKŁAD: *Nearables* czyli Estimote iBeacons

Produkt krakowskiej firmy Estimote (aktualnie działającej w ośrodkach Kraków - Nowy York - San Francisco) to specjalne sensory *(becons)* do tworzenia sieci IoT.

Ilustracja 40: Estimote iBeacon stickers[58]

[58] Ilustracje i przykłady pochodzą ze strony internetowej producenta: http://estimote.com (dostęp 10.08.2014)

Beacony zbudowane są z niewielkich rozmiarów, tanich układów scalonych (będących tak naprawdę mikrokomputerem z 32-bitowym procesorem ARM Cortex M0 z 256kB pamięci masowej), z którymi można się komunikować przy użyciu energooszczędnej technologii radiowej Bluetooth 4.0 Smart *(Bluetooth Low Energy,* który jest zupełnie nowym standardem, różniącym się znacząco w stosunku do "starego" BT). *Boje* emitują stały sygnał, który może być odbierany i interpretowany przez dedykowane aplikacje w komputerach, tabletach i telefonach komórkowych. Twórcy wraz z urządzeniami udostępniają API, dzięki któremu programiści mogą stworzyć w zasadzie dowolne oprogramowanie wykorzystujące informacje z sensorów pozwalając na operowanie urządzeń komunikacyjnych w fizycznym kontekście.

Ilustracja 41: *boja* Estimote składa się z układu scalonego z emiterem, źródła zasiania i eleganckiej obudowy.

Interesującym przypadkiem wykorzystania rozwiązania Estimote iBeacons jest przeprowadzone przez agencję interaktywną Prophets wdrożenie w Rubens House Museum w Antwerpii. Wyposażonego w urządzanie z systemem Android lub iOS zwiedzającego po wystawach "oprowadzają" inteligentne moduły, które *potrafią* rozpoznać w jakiej odległości od poszczególnego eksponatu znajduje się użytkownik, w którą stronę jest obrócony, jak długo pozostaje w bezpośredniej bliskości obiektu i w zależności od tego zaproponować zapoznanie się z detalami obrazu, zaprezentować historię powstania rzeźby czy wyjaśnić funkcję danego utensylium.

Ilustracja 42: Estimote w Rubens House Museum

Oprogramowanie tworzone na potrzeby IoT ma swoją specyfikę, również ze względu na platformę, na którą jest tworzone. Dla przykładu prosty kod źródłowy dla iOS (Objective-C) wygląda następująco:

```
import Estimote

let estimote = Estimote(appID: "myapp", secret: "contextisking")
estimote.startDiscoveringNearables(type:
Estimote.NearableType.ArtPiece)

func estimoteDidDiscoverNearables(nearables: [Estimote.Nearable]!)
{
    let artThatMayInterestThisUser = nearables.filter {
      $0.isInMotion(forTheLast: 5.seconds)
    }
    if (estimote.user.isReturningVisitor(within: 1.month)) {
        // note visitor to the changed exhibit
    }
}
```
[59]

Jak widzimy z podkreślonych na niebiesko wyrażeń, tak zaprogramowany beacon będzie w stanie *rozpoznać* powracającego zwiedzającego i zaproponować mu za pośrednictwem aplikacji zwrócenie uwagi na zmieniony eksponat oraz skierować jego uwagę na potencjalne interesujące obiekty tego rodzaju.

PRZYKŁAD: *Smart City* i Ifinity Beacons

Innym przykładem podobnego projektu jest warszawski startup Ifinity. Również tutaj mamy do czynienia z inteligentnymi (choć może właściwiej było by nazywać je *perceptywnymi)* beaconami wyposażonymi w szereg czujników, dzięki którym mogą dostarczać informacji o jakości powietrza, wilgotności, naświetleniu, zaburzeniach pola

[59] zmodyfikowany kod źródłowy z: http://estimote.com (dostęp 11.08.2014)

magnetycznego oraz - oczywiście - wykrywać obecność urządzeń mobilnych wyposażonych w odpowiednie oprogramowanie i komunikować się z nimi.

Ilustracja 43: Ifinity Virgo Beacon[60]

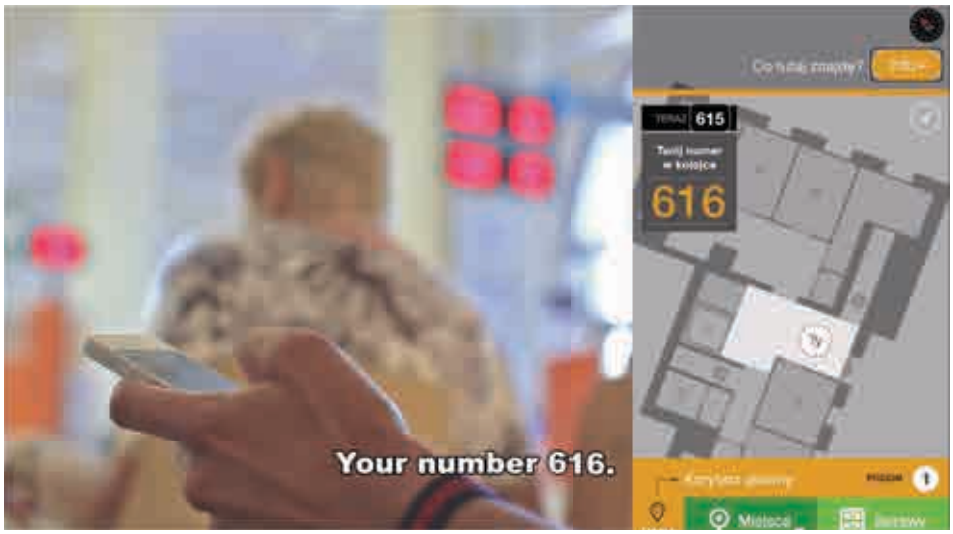
Projekt Ifinity rozwija się bardziej niż Estimote w stronę nie tyle nadania Sieciowej *osobowości* przedmiotom życia codziennego co aktywowania informacyjnego przestrzeni miejskiej (choć oba opisywane projekty mogą się sprawdzić w różnych dziedzinach). Podstawowe przypadki użycia obejmują tu:

- lokalizowanie ruchomych obiektów w przestrzeni miejskiej w czasie rzeczywistym;
- zwiększanie dostępności pewnych obszarów dla osób niepełnosprawnych;
- mikrolokalizację w biurach, urzędach, muzeach, targach;
- zarządzanie dostępnością miejsc na parkingach, w bibliotekach, trasach do zwiedzania;
- płatności kontekstowe;
- analiza przepustowości węzłowych punktów komunikacyjnych;
- inwentaryzacja cennych zasobów;

etc.

[60] materiały pochodzą ze strony: http://getifinity.com (dostęp 12.08.2014)

Ciekawym *case study* jest tu projekt *Virtual Warsaw* realizowany przy współpracy z władzami tego miasta. W wersji testowej użytkownik może korzystać z mikronawigacji przy poruszaniu się do budynku, sprawdzić gdzie załatwia się dany rodzaj sprawy, pobrać numer w kolejce, zapoznać się z ogłoszeniami i tablicami informacyjnymi.



Ilustracja 44: Wizyta w urzędzie wspomagana aplikacją *Virtual Warsaw* działającą w oparciu o technologię *Ifinity.*

Szacuje się, że do 2020 roku do Internetu w ten czy inny sposób będzie podłączone 50 miliardów urządzeń[61]. Jest to gigantyczne wyzwanie dla twórców oprogramowania i architektów technologii - połączenie do Sieci nie oznacza bowiem jeszcze pełnej wymiany informacji, a jedynie pewien potencjał w tym zakresie. Wydaje się, że w najbliższych latach będziemy obserwować lawinowy wzrost różnego rodzaju technologii i idei związanych z ubiquitous computing. Tym bardziej, że zjawisko rozwija się nie tylko w środowiskach startupowych i dużych firmach ale również za sprawą ruchów DIY *(Do It Yourself), hardware hackerów, media labów* itp. Ułatwia to coraz szerszy dostęp do tanich i otwartych podzespołów i platform w rodzaju Arduino czy RaspberryPi (o których mowa więcej w rozdziale T2). Społeczności ludzi kreatywnych pełnią istotną rolę w rozwoju, a przede wszystkim adaptacji nowych technologii. Szerzej tym zjawiskom przyjrzymy się w rozdziale Δ3 niniejszej pracy traktującym o partycypacyjności.

[61] źródło: http://share.cisco.com/internet-of-things.html (dostęp 12.08.2014)

Interfejs między częściami 2:
Od A do Δ: *There is no software.*

"Ten trwający triumf oprogramowania jest dziwnym odwróceniem dowodu Turinga, że nie może istnieć obliczalny matematycznie problem, którego nie potrafiłaby rozwiązać prosta maszyna. Zamiast tego, hipoteza Churcha-Turinga poprzez zrównanie fizycznego hardwaru z algorytmami stworzonymi na potrzeby komputacji, w końcu doprowadziła do pozbycia się (z powszechnej świadomości - dop. J.K.A.) samego hardwaru. W rezultacie, oprogramowanie zajęło to puste miejsce i skorzystało ze swojej niejednoznaczności."

(Kittler, 1997, p. 151)

Zmarły w 2011 roku badacz nowych mediów Friedrich Kittler, to postać niezwykła. Jako jeden z nielicznych medioznawców stworzył on "szkołę" o niezwykle silnej, choć i bardzo nieraz kontrowersyjnej tożsamości teoretycznej. Z jednej strony jego wywody charakteryzują się inżynierską precyzją (i zaborczością), z drugiej nie stroni on od radykalnych eksperymentów myślowych i dalekosiężnych metafor. Esej Kittlera pt. *There is no software* może stanowić interesujące podsumowanie naszych dotychczasowych analitycznych rozważań - kieruje bowiem uwagę na zagadnienie specyficznego wieloaspektowego sposobu istnienia kodu, która to intuicja towarzyszy nam od początku tej pracy. Na czym więc polega zróżnicowanie modusów istnienia oprogramowania?

Pierwszy z nich wynika z wielokrotnie już przytaczanego tu i analizowanego modelu teoretycznego działania komputerów: to Uniwersalna Maszyna Turinga jako koncept teoretyczny. "Potrafi" ona jak wiemy dokonać odczytu i zapisu na "taśmie" oraz przemieścić się. Tak prosty konstrukt, odpowiednio *oprogramowany* stanowi ekwiwalent każdej funkcji obliczalnej.

Kolejny poziom tak naprawdę składa się z wielu pośrednich płaszczyzn, które objaśnia autor za pomocą działania w edytorze tekstu (konkretnie jest to WordPerfect ale fakt ten nie ma większego znaczenia). Otóż uruchamiając program zostajemy poddani działaniu silnej metafory pochodzącej ze świata realnego, prezentuje się nam urządzenie obliczeniowe w funkcji *maszyny do pisania*. Tak wygląda i działa to, co widzimy na ekranie, podczas gdy tak naprawdę - powiada Kittler - czynność pisania na komputerze nie jest po prostu pisaniem, takim jakie odbywa się za pomocą pióra czy maszyny z czcionkami. Pisanie w edytorze to przetwarzanie danych, obliczenia, implementacja Maszyny Turinga.

Oprogramowanie *maskuje* przed nami ten fakt podczas gdy WordPerfect potrzebuje do swojego działania systemu operacyjnego zawiadującego pracą jednostki centralnej, peryferiów, pamięci. System operacyjny potrzebuje natomiast do swojego rozruchu BIOS'a, podstawowego systemu wejścia/wyjścia, który przetworzy podstawowe impulsy elektryczne i zadba o właściwe środowisko, na którym będzie mógł wykonać swoją pracę system.

"Co do zasady, ten rodzaj zstąpienia od oprogramowania do sprzętu, od wyższych do niższych poziomów obserwacji powinien być kontynuowany przez coraz większe stopnie powiększenia. Wszystkie operacje realizowane przez kod, pomimo takich metaforycznych określeń jak wywołanie czy powrót redukują się do absolutnie podstawowych manipulacji ciągami znaków, co oznacza - obawiam się - do *elementów oznaczanych przez różnice stanów elektrycznych*." (ibid. p.150)

Graficzne interfejsy jawią się więc jako konstrukty mające za zadanie anihilować *maszynowość* komputera w oczach użytkownika. Spostrzeżenie to zyskuje właściwą wagę w momencie kiedy uświadomimy sobie pozornie dość trywialną prawdę, że nasze algorytmy nie są *tak naprawdę* realizowane przez Uniwersalną Maszynę Turinga lecz przez jej ograniczoną w czasie i przestrzeni, a więc i zasobach implementację. Rzeczywistość przetwarzana przez komputery musi więc być odpowiednio:

zredukowana,
skompresowana,
przetworzona,
zdyskretyzowana
i skwantyfikowana,

a to już z całą pewnością nie są całkowicie neutralne procesy.

"W ten sposób zamiast podążać tropem hipotezy Churcha-Turinga i »wstrzykiwać algorytmiczny behawior w zachowania świata fizycznego (...)«[62] musimy raczej przyjrzeć się temu, co zostało samo nazwane »ceną za programowalność«. Ta fundamentalna

[62] Kittler cytuje tu siebie samego z innego tekstu pt. *The Mechanized Philosopher*

właściwość bycia programowalnym nie ma, wedle wszelkich przesłanek, nic wspólnego z oprogramowaniem; jest wyłączną właściwością hardwaru (...)"

(ibid. p. 153)

W charakterystyczny dla siebie sposób Kittler staje w obronie poglądu nieintuicyjnego. Komputer jest maszyną, kod jest kognitywną protezą, modelem lub matematyczną abstrakcją której - jak pisze filozof - *eksplozja* w postaci wysokopoziomowych języków programowania spowodowała *implozję* hardwaru.

Jak zauważyliśmy w poprzednim rozdziale poświęconym kwestiom ubicompu i IoT kwestia hardwaru staje się coraz bardziej znacząca we współczesnym świecie. Komputery stają się nie tylko coraz tańsze i wydajniejsze ale zmieniają swoją zewnętrzną postać. Stają się częściami garderoby, zegarkami, okularami, brelokami do kluczy, przedmiotami codziennego użytku a nawet fragmentami naszego organizmu dzięki cybernetycznym wszczepom. Uwagi Kittlera pozostają jednak w mocy. Dla zwykłego "użytkownika rzeczywistości" komputery *wbudowane* w otaczające obiekty pozostają *przeźroczyste*. Większość nie zwróci uwagi na to *dlaczego* po wejściu do sklepu zostają przywitani SMS-em z imiennym nagłówkiem i propozycją skorzystania z oferty produktów w swoim rozmiarze i swoim ulubionym kolorze. Technologia *chce* pozostawać niewidzialna. Dlatego tak ważne wydaje się, często z pozoru nieco zbyt techniczne, wyjaśnianie sposobów i mechaniki działania pewnych obiektów medialnych. W tej roli prezentowana była druga część niniejszej dysertacji.

Spróbujmy pamiętać o tej krytycznej perspektywie kiedy w kolejnej części pracy oderwiemy się nieco od kwestii związanych z implementacją oraz średnio i niskopoziomowym działaniem oprogramowania by przyjrzeć się **dynamikom** przez nie implikowanym.

Część 3: Dynamiki (Δυναμική)

Δυναμισ. Dynamis. „Termin ten, który może być poprawnie tłumaczony jako *możność,* stopniowo nabierał wiele różnych znaczeń (...) w *Metafizyce* (Arystotelesa - dop. J.K.A.) czytamy ≫(1) Możnością nazywa się *zasadę ruchu lub zmiany w czymś drugim*, albo w czymś jako drugim. Na przykład sztuka budowania jest taką możnością, która nie występuje w tym co budowane, natomiast możność leczenia może posiadać i ten, kto podlega leczeniu, ale nie jako leczący. (2) A więc *możność jest w ogóle zasadą ruchu lub zmiany w czymś drugim*, albo w czymś drugim jako drugim, jak również jest zasadą zmiany powodowanej przez coś drugiego jako drugiego. (...) ≪.”

(Reale 2005, p. 63)

Δυναμική 1: Oprogramowanie jako kultura.

"Mowa tu o procedurach, dzięki którym wytwory kodu binarnego pozostają na etapie nieustannej gotowości na polecenia z centrum dystrybucji i zarządzania - mają sposobność aby uzupełnić brakujące sekwencje, zdalnie naprawić pojawiające się błędy czy poszerzyć swoją funkcjonalność. W tym sensie miękki, programistyczny poziom nowych mediów jest otwarty i podatny na zmiany, nigdy ostatecznie nie ukończony, stale oczekujący na nowe dyspozycje i reorganizacje."

(Celiński, 2010, p. 53)

"Komputery, jak inne media, są maszynami metaforycznymi: zależą od nich i podtrzymują je. Co jednak bardziej niezwykłe - ze względu na swój status ≫uniwersalnych maszyn≪ - stały się same metaforami metafor (...)

Komputery stały się metaforami dla umysłu, kultury, społeczeństwa, ciała, wpływając na sposoby w jakie doświadczamy i konstruujemy pojęcie ≫rzeczywistej≪ przestrzeni: od zaprogramowanych umysłów działających na konkretnie zbudowanych mózgach po reprogramowalną kulturę w opozycji do opartej na ścisłych prawach natury, od sieci neuronowych do programów genetycznych (...)

Ponieważ komputery są postrzegane jako uniwersalne maszyny, stały się metaforami dla samej metafory: uosabiają logikę zastępowania, ledwie widoczny konceptualny system, który porządkuje i rozstraja. Metafora pochodzi od greckich terminów *meta* (zmiana) i *phor* (dźwiganie): to transfer, który transformuje."

(Chun, 2011, pp. 55-56)

Techniczna definicja oprogramowania mówi o jego konstrukcji ("informacje w postaci zestawu instrukcji zaimplementowanych interfejsów i zintegrowanych danych przeznaczonych dla komputera do realizacji wyznaczonych celów"[63]) i jego celach ("celem oprogramowania jest przetwarzanie danych w określonym przez twórcę zakresie"[64]). Niewiele natomiast da się z niej wyprowadzić wniosków na temat społecznych czy kulturotwórczych sposobów działania oprogramowania, a także w kwestii źródeł jego takiej a nie innej budowy. Na tą drugą kwestię rzuciliśmy nieco światła w poprzednich rozdziałach, przyjrzyjmy się więc problemowi *funkcjonowania* softwaru w kontekście jawnych i ukrytych *znaczeń* jakie - działając - generuje.

Lew Manovich w swojej książce *Software Takes Command* nazywa komputer metamedium:

"Jest ono unikatowe po wieloma względami. Jeden z nich dyskutowałem już szczegółowo - potrafi ono reprezentować większość innych mediów poszerzając je o szereg nowych właściwości. (...) Nowe metamedium jest »aktywne - potrafi reagować na zapytania i eksperymenty - więc komunikacja może angażować uczenie się w obu kierunkach«."

(Manovich, 2006, p. 102)

Autor zwraca uwagę również na fakt, że oprogramowanie będąc w założeniu narzędziem rozwiązywania problemów jest też sposobem zachowywania pamięci (o czym była mowa w jednym z poprzednich rozdziałów) a także *generatorem* nowych mediów i niespotykanych dotąd sposobów działania medialnego. To bardzo potężna właściwość, która wyznacza software jako praktycznie nieograniczone narzędzie symulacji istniejącej i kreowania nowej rzeczywistości medialnej.

---

[63] źródłem fragmentów definicji jest Wikipedia, która stanowi dobre odniesienie, jako że zawiera hasła wypracowane zbiorowo przez pewną społeczność (często mocno zinformatyzowaną) i może być świadectwem pewnego konsensusu. Vide: http://pl.wikipedia.org/wiki/Oprogramowanie (dostęp 1.06.2014)

[64] ibid.

Metamedialna charakterystyka kodu objawia się również we (wspominanym już) operacyjnym potencjale krytycznym tego fenomenu. Matthew Fuller pisze:

"Są dwa kluczowe tryby. Przez użycie dowodów prezentowanych przez normalizowane oprogramowanie do konstruowania struktur obiektów, protokołów, wyrażeń, dynamik i sekwencji interakcji, które pozwalają na zamanifestowanie warunków prawdziwości (...) [oraz] w różnych instancjach oprogramowania, które działa jak zupełnie normalne aplikacje ale zostało fundamentalnie zaburzone w celu ujawnienia ukrytej pod powierzchnią konstrukcji użytkownika, sposobów przetwarzania danych, transdukcji i źródłowych procesów interfejsu". (Matthew Fuller, 2003, p. 11)

co przekłada się na specyficzny wymiar *spekulatywności* w odniesieniu do własnej charakterystyki. Oprogramowanie - uważa autor cytowanego tekstu - może otwierać przestrzeń do reinterpretacji siebie za pomocą swoich własnych środków. Software jest procesualnym nośnikiem pewnego rodzaju operacyjnych sensów, co oznacza, że ma *wpisane* w swoją konstrukcje polityki działania, hierarchie i modele kulturowe. W rezultacie nowe sposoby tworzenia, przetwarzania, dystrybucji i konsumpcji obiektów kulturowych nie są skutkiem jedynie faktu ich *digitalizacji* lecz przede wszystkim ich kształt i charakter determinowany jest działaniem oprogramowania, a więc systemów opracyjnych, softwaru narzędziowego do tworzenia i edycji, kodeków, formatów, baz danych i dalej - języków programowania, których użyto do ich stworzenia, konceptualnych architektur na jakich je zbudowano.

"Krótko mówiąc: nowe media stają się softwarem" (Manovich, op. cit. p. 156), zaś cyfrowa kultura jest nim zdeterminowana.

Δυναμική 2: Paradygmat rozproszony.
Sieć, protokoły komunikacyjne.

„Internet stanowi tkankę naszego życia. Gdyby technologię przetwarzania informacji uznać za dzisiejszy odpowiednik tego, czym była elektryczność w erze przemysłowej, to Internet z uwagi na jego zdolność do przekazywania siły informacji na wszystkie sfery ludzkiej działalności, można by porównać zarówno do sieci energetycznej, jak i silnika elektrycznego. Ponadto tak, jak nowe techniki produkcji i dystrybucji energii elektrycznej uczyniły z fabryk i dużych firm organizacyjne podstawy społeczeństwa industrialnego, tak Internet stanowi techniczną bazę dla struktury organizacyjnej epoki informacji – sieci”

(Castells, 2003, p. 11)

"Podczas gdy my jesteśmy w nocy martwi dla świata, sieci maszyn bezdźwięcznie i bezustannie wymieniają dane. Monitorują, kontrolują i łączą się ze światem za pośrednictwem elektronicznych czujników, aktualizują listy i bazy danych, obliczają i przetwarzają swoje modele w celu wyprodukowania raportów, projekcji i ostrzeżeń. W wibrujących konstelacjach danych, nadzorują one i stabilizują życie codzienne jednostek, grup i organizacji, posiadają informacje na temat wzorców kryminalnych, nietypowych zachowań i odstępstw w zaprogramowanych statystycznie modelach. W czasie naszej dziennej aktywności potężna ilość maszyn otwiera i zamyka drzwi, zawiaduje sygnalizacją świetlną zmieniając kolory z czerwonego na zielony i z powrotem, monitoruje i autoryzuje (lub nie) nasze karty płatnicze i generalnie utrzymuje świat w ruchu."

(Berry, 2011b, p. 1)

Nie podlega wątpliwości fakt, że Sieć stanowi najpotężniejszy motor dla dynamik komunikacyjnych, dzisiejszych czasów, a co za tym również jeden z najważniejszych silników przemian kulturowych. Istnieje cała gama ram pojęciowych w jakie można ujmować problematykę z wiązaną z Internetem, ogromna jest również wielość teorii mogących posłużyć do wyjaśnienia procesów w nim zachodzących. Dla nas, oczywiście, najważniejsza jest perspektywa refleksji nad oprogramowaniem stąd istotne z tej perspektywy tematy obejmować mogą: problem systemów pojęciowych i idei leżących u podstaw paradygmatu sieciowego (onto-topologia, ANT, reticulum); software regulujący funkcjonowanie obiegów sieciowych (protokoły); a także obecność form artystycznego wyrazu będących swoistym badaniem *wytrzymałości* i granic medium (net.art).

Nie będziemy się tym razem (dla odmiany) wikłać w perspektywę historyczną, ani rozważania dotyczące konkretnych konstrukcji architektonicznych czy rozwiązań technologicznych pozostając na poziomie czystych dynamik systemu, struktur konceptualnych i obiektów w nie włączonych.

Onto-topologia sieci

Filozofia dostarcza licznych tropów metafizycznych, które można uczynić filarami podtrzymującymi gmach teoretycznych rozważań na temat sieci i sieciowości. Jednym z bardziej dla nas interesujących ujęć jest propozycja silnie osadzonej w paradygmacie leibniziańskim *onto-topologii,* której założenia zostały sformułowane m.in. przez Leslie Jaye Kavanaugh.

Podstawowym zadaniem metafizyki jest oczywiście próba odpowiedzi na pytanie o istnienie. Może być ono zadane na wiele różnych sposobów i dotyczyć różnych aspektów tego zagadnienia. Zdaniem filozofki jednak najważniejszym wynikiem pracy intelektualnej ontologa nie jest sama udzielona odpowiedź (lub jej brak) lecz:

"Być może, stwierdzam, jest (metafizyka - dop. J.K.A.) przede wszystkim tym wielkim dążeniem do konstruowania, do zawarcia, do ograniczenia. Jeśli tak to czy jej konstrukty są opcjonalne? Czy mogą zostać zaproponowane inne struktury, inaczej skonfigurowane a jednak bardziej użyteczne i znaczące od współczesnych?" (Kavanaugh, 2007, p. 26).

Topologia może być zdefiniowana - pisze autorka - jako relacja pomiędzy połączonymi elementami systemu. W takim wypadku *onto-topologia* jako metoda to działanie polegające na czynieniu jawnymi struktur ontologicznych jakie podbudowują projekt Zachodniej metafizyki: "onto-topologia - logos »sytuacji« bycia - jest po prostu pytaniem: »*Gdzie jest* bycie?«." (ibid. p. 27).

Elementy *Actor-Network Theory*.

Zostawmy na chwilę *twardą* metafizykę i jej problemy (za chwilę do nich wrócimy) i zajmijmy się tymczasem innym z podstawowych problemów myślenia o sieciach - kwestii związanych z funkcjonowaniem układów (społecznych) o dużej złożoności.

Ciekawe ujęcie tej problematyki proponuje Betina Szkudlarek w swojej pracy poświęconej tematyce ontologizowania relacji w ujęciu ANT (*Actor-Network Theory)* czyli popularnej *teorii aktora-sieci* (Szkudlarek, 2007). Jest to bardzo dogodny dla nas model, gdyż pozwala on na ujęcie dynamicznego charakteru relacji między obiektami bez konieczności uwzględniania ich podmiotowo-ludzkiego charakteru i równoczesnego skazywania nas na czysto techniczne rozważania z dziedziny telekomunikacji. Punktem wyjścia koncepcji Szkudlarek jest paradygmat jakościowy w socjologii organizacji ze swoimi licznymi modusami - oferują one zbiorczo instrumentarium pozwalające na konstruowanie, rekonstruowanie i dekonstruowanie organizacyjnych rzeczywistości. W oparciu o prace głównie Johna Lawa, należące do tak zwanej *późnej* ANT (Law, 1999) dają się zaprezentować trzy dystynktywne ujęcia sieciowości. Każde z nich ma swoje istotne momenty oraz problematyzuje inny aspekt tego samego zjawiska:

- *przestrzenność sieci (network spatiality):* "może być rozumiana jako system, gdzie relacje pomiędzy różnymi elementami układu są konstytuantami tego systemu a obiekty są efektami stabilnej tablicy lub sieci relacji" (Szkudlarek, 2007, p. 194)

- *przestrzenność obszaru (region spatiality):* to drugi rodzaj topologii społecznej, który może nas interesować. Obszar nie jest tu rozumiany jako coś przestrzennego w sensie eklidesowym czy trójwymiarowym. Region to, cytuje Johna Urry'ego autorka, przestrzeń "w której obiekty są układane razem, a granice są nakreślane wokół każdego z obszarowych elementów" (za: ibid, p. 195). W przeciwieństwie do sieci - zauważa Szkudlarek - obiekty w ramach topologii obszarowych nie muszą być bezpośrednio połączone ani interkonektywne.

- *przestrzenność płynna (fluid spatiality)*: w topologii płynnej, jak sama nazwa wskazuje, nie są wyznaczone tak wyraziste granice jak w przypadku ujęcia obszarowego; nie są również określone równie jednoznaczne relacje jak w topologii sieci: "tu ograniczenia i połączenia są rozmyte a nawet nierozróżnialne. Obiekt istnieje ze względu na daną możliwość powstania, rozwoju i zmiany - i ta mobilność, niestabilność i falowanie są warunkami, w ramach których obiekt nieustannie zmienia swoje kształty. Te zmiany są jednak raczej łagodne i subtelne, niż gwałtowne i brutalne; nowe elementy mogą się przyłączać, dotychczasowe odrywać" (ibid. p. 196)

Koncepcja "Reticulum".

Wróćmy tymczasem do metafizyki. Dysponujemy już pojęciem *onto-topologii* oraz kilkoma możliwymi wizjami tego czym jest sama topologia w ujęciu relacji i dynamik między różnymi, niekoniecznie ludzkimi podmiotami (aktorami, agentami). Uprzednio wspomniana leibniziańska inspiracja całej koncepcji rozważania Sieci w kontekście dynamicznym zasadza się na architektonice *reticulum.* Słowo to pochodzi z języka łacińskiego i oznacza *siatkę* w rozumieniu matematyczno-geograficznym (czyli bliżej angielskiego *mesh* niż *web* lub *network).* Za pomocą tej metafory wspominana już Leslie Kavanaugh wyjaśnia konstrukcje leżące jej zdaniem u podstaw europejskiego, nowożytnego projektu metafizycznego. Nie wdając się w fascynujące, lecz niezwykle hermetyczne rozważania na temat *monadologii* zwróćmy uwagę na to jak autorka rekonstruuje działanie systemu metafizycznego Leibniza:

"W reticulum, każda monadyczna jednostka, jako autonomiczny lecz interkonektywny byt, wyraża i jest wyrażana; każda monada transformuje i jest odbiorcą wpływu transformacji innych monad; każda monada sięga z pożądaniem (l'appetit) i swobodnie przyłącza do siebie inne monady, zmienia się wewnętrznie, rozprasza i wtedy bierze udział w formowaniu innych podrzędnych agregatów monadycznej substacji. W przeciwieństwie do wczesnych koncepcji Leibniza, że »monady nie mają okien«, subtelniejszy późny model zakłada aktywne monady w złożonej substancji co przekłada się na responsywny układ"

(Kavanaugh, 2007, p. 29)

Oczywiście nie jest naszą rzeczą na kartach tej dysertacji rozważać kwestie sposobów istnienia substancji u nowożytnych filozofów - to co interesujące wynika z konsekwencji tak postawionego stanu systemu monadycznego. Filozofka nazywa bowiem leibnizowski koncept specyficzną architektoniką, która jako jedna z pierwszych lub wręcz pierwsza buduje wizję złożonego systemu opartego na sieci (*siatce): reticulum.* W strukturze tej - przynoszącej historycznie nową niehierarchiczną, dynamiczną monadologię - *istnienie* oznacza funkcjonowanie w międzysubstancjalnej złożoności pozostających w relacji obiektów. Jak mówi autorka: taka "struktura spłaszcza się do radykalnej immanencji".

Po raz kolejny okazuje się, że Gottlieb Willhelm Leibniz wyprzedził intelektualnie swoje czasy i opisał strukturalną dynamikę współczesnej sieci na kilkaset lat przed jej powstaniem.

Protokoły.

Powróćmy zatem do teraźniejszości i pamiętając o wizji sieci-*reticulum* oraz teorii aktora-sieci (szczególnie w zrekapitulowanym powyżej aspekcie) sięgnijmy do problematyki Internetu czyli pewnego obiektu technologicznego, składającego się z komputerów (a więc jak wiemy - maszyny do symulowania maszyn) charakteryzującego się specyficzną strukturą (sieć-*web*) i regułami opisującymi relacje między jej elementami (*protokoły*). Te ostatnie są najbardziej interesujące z perspektywy *software studies*.

Aleksander Galloway zajmując się nimi w jednej ze swoich książek pisze:

"Protokół nie jest nowym słowem. Przed swoim zastosowaniem w informatyce protokół odnosił się do każdego rodzaju poprawnego zachowania w ramach specyficznego systemu konwencji (...) Lecz, z najdejściem cyfrowych technologii obliczeniowych pojęcie zyskało nico inne znaczenie. Teraz, protokoły odnoszą się konkretnie do standardów implementacji konkretnych technologii. Podobnie do swoich dyplomatycznych poprzedników, protokoły komputerowe ustanawiają podstawowe punkty konieczne do działania na uzgodnionym, standardowym gruncie." (Alexander Galloway, 2004, p. 7)

Jak zwraca uwagę autor - reguły konstruują zestawy możliwych (lub lepiej - tolerowalnych czy interpretowalnych) zachowań w złożonych, niejednoznacznych systemach. Protokół stanowi więc regulację dotyczącą pożądanego zachowania w danym środowisku. Dla nas najbardziej istotnym jest, że protokoły działają na poziomie kodu - "kodują pakiety informacji aby mogły być przesłane; kodują dokumenty aby mogły być efektywnie parsowane; kodują komunikację aby lokale urządzenia mogły efektywnie komunikować się z obcymi. Protokoły są wysoce formalne (...)" (ibid.)

Przykładem jednego z najbardziej powszechnych protokołów jest TCP/IP *(Transmission Control Protocol/Internet Protocol)*, który opisuje części składowe (dalsze protokoły) w poziomach dostępu do Sieci:

- warstwę aplikacji *(process layer)*: serwery WWW, przeglądarki internetowe etc. Zawiera dalsze protokoły służące do obsługi stron WWW (HTTP - *Hypertext Transfer Protocol),* przesyłu plików (FTP - *File Transfer Protocol),* adresowania domen (DNS - *Domain Name Server)* i wiele innych;

- warstwę transportową *(host-to-host layer):* realizuje komunikację poprzez przydzielanie transferów danych do portów zapewniając bezkonfliktowość transmisji;

- warstwę Internetu *(Internet protocol layer):* odpowiada za *interpretowanie* adresów IP w Sieci, routing połączeń (tutaj działa *topologia* Sieci);

- warstwę dostępu do sieci *(network access layer):* to najniższa warstwa fizyczna protokołu. Dane odpowiednio sformatowane i zakodowane muszą zostać przez pewne urządzenia (modemy, karty sieciowe) przesłane do Sieci.

TCP/IP jest obecny niemal w każdym komputerze (czy właściwie: w systemie operacyjnym komputera) stanowiąc jedno z najpopularniejszych rozwiązań komunikacyjnych na świecie.

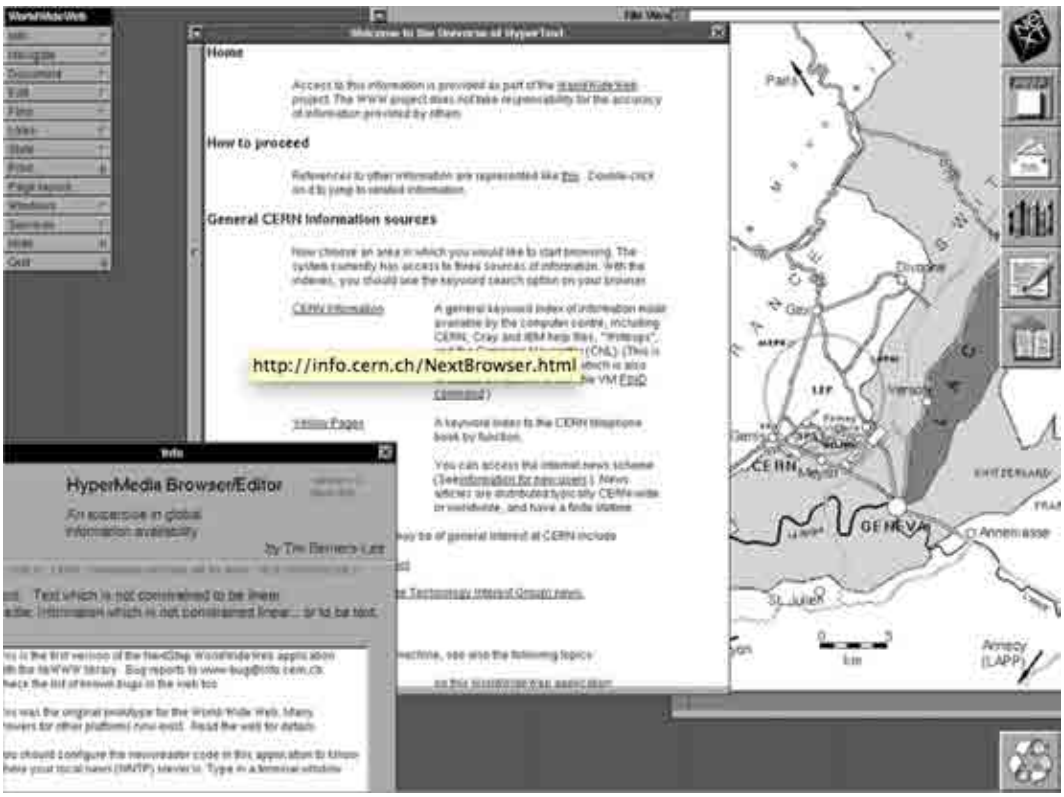
Internetowe protokoły mają budowę kompozytową (mogą się w nich znajdować inne, a z kolei w tych jeszcze inne) i działają w modelu zdecentralizowanym (ARPA opracowywała Sieć jako rozwiązanie mające zapewnić komunikację na wypadek ataku jądrowego - stąd "odcięcie" punktu węzłowego czy dużego *hosta* nie mogło niweczyć pracy systemu). Dystrybucyjny charakter Internetu jest czymś bardzo ważnym nie tylko z perspektywy praktycznej ale i teoretycznej. Oto bowiem globalna Sieć pełna jest różnego rodzaju obiektów, podmiotów, działań, idei, które w sposób konieczny muszą się ze sobą komunikować - inaczej zostaną wyłączone poza margines tej rzeczywistości. Część z nich przestanie nawet istnieć całkowicie. Jeśli powrócimy teraz do opisywanych na początku: systemu metafizycznego (czy właściwie struktury stanowiącej jego szkielet) zdamy sobie sprawę, że dokładnie o czymś takim mówi Leibniz i to rekonstruuje Kavanaugh - swoisty Internet *substancji* prostych transformujących się pod wpływem własnych relacji i budujących w ten sposób sieć, tolpologię, reticulum, która jednocześnie jest podstawą bytową, warunkiem aktywności jak i przestrzenią, dzięki której obiekty mogą "orientować się" w wzajemnym położeniu. Ciekawy to model metafizyczny obejmujący jednocześnie hardwarowy i softwarowy charakter Internetu, czyniący dodatkowo z twórczej aktywności

użytkowników, transferu idei, główną oś konstrukcyjną. Takie szerokie spojrzenie będzie nam potrzebne kiedy przyjrzymy się zjawisku transdukcji i procesom, które są nim związane.

Net art - HTML art

W Sieci istnieje nieprzebrane bogactwo obiektów, które mogą (lub chcą) być kwalifikowane jako Sztuka Internetu czy Sztuka Sieci. Oczywiście ich włączenie lub dyskwalifikacja z tego zakresu zależy od przyjętych ram estetycznych i modelu sądzenia o przynależności do świata sztuki. Nas przede wszystkim interesują dwa uwarunkowania tworów artystycznych, mających "miejsce" w Internecie: ich wymiar softwarowy oraz natywność w cyberprzestrzeni (nie wystarczy więc, że dzieło *jest* w Internecie, musi być z nim nierozerwalnie związane). Wobec tak postawionych kryteriów możemy przyjąć, że jednym z głównych nurtów net artu jest ekspresja na bazie stron WWW - HTML art.

"Wynalazcą" a więc i pierwszym autorem WWW jest Tim Berners-Lee, który stworzył swoją pierwszą stronę internetową w 1990 roku, kiedy gdy pracował w CERN. Język opisu dokumentu hipertekstowego oparty został oparty na SGML *(Standard Generalized Markup Language).* Przy udziale Roberta Caillau opracowana została architektura *sieci* dokumetów *hipertekstowych* odczytywanych przez specjalną *przeglądarkę* działającą w modelu klient-serwer. Ostatnim etapem, po uwolnieniu rozwiązania i wykazania jego skuteczności w małej sieci było skonstruowanie systemu lokalizowania dokumentów wg. nazw (adresów) URL *(Uniform Resource Locator)*, dokumentacja pełnego języka opisu hipertekstowych dokumentów HTML *(Hypertext Markup Language)* i opracowanie protokołu wymiany informacji dla WWW - HTTP *(Hypertext Transfer Protocol).*



Ilustracja 45: Strona internetowa 1990 r.
Screenshot interfejsu komputera NExT
(który był też pierwszym serwerem www)

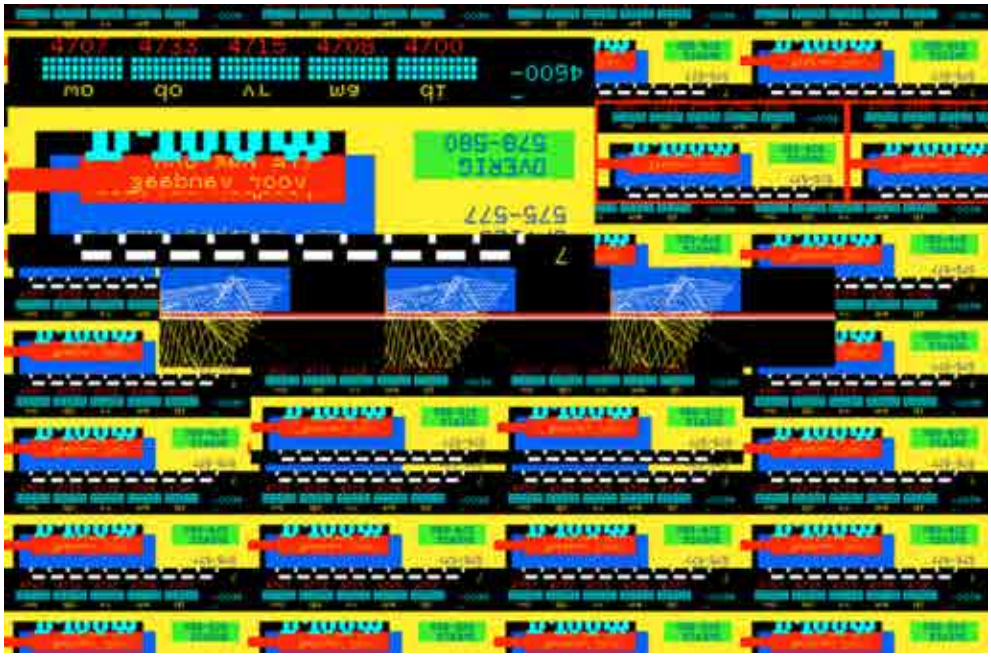
Tworzenie stron internetowych przy użyciu HTML'a to rzecz bardzo prosta. Nie jest on nawet językiem programowania lecz jedynie *opisu* hipertekstu. To znaczy, że kod źródłowy jest nie tyle *wykonywany* przez przeglądarki co *interpretowany* przez ich silniki i renderowany zgodnie z własnym programem. Oczywiście sprawy komplikują się kiedy czysty HTML wzbogacony zostanie różnego rodzaju językami skryptowymi takimi jak JavaScript czy PHP.

Strony internetowe od samego początku (jak deklaruje Berners-Lee) tworzone były z myślą o wygodnym odczycie w postaci dokumentu, kod źródłowy nie był przeznaczony - podobnie jak w oprogramowaniu - do czytania przez odbiorcę (choć z przyczyn technicznych jest znacznie bardziej łatwiej, generalnie da się czytać źródła większości stron internetowych o ile twórca nie zadał sobie specjalnie trudu aby go "zamazać" - uczynić bardzo trudno czytelnym dla człowieka).

Najciekawsze projekty z dziedziny net artu to jak się wydaje te, które w jakiś sposób dekonstruują metaforę dokumentu, na której oparte są interfejsy WWW lub też takie, które problematyzują przepływ danych i infrastrukturę samej Sieci.

Przykłady: *Web projects* Jodi oraz Web Stalker I/O/D.

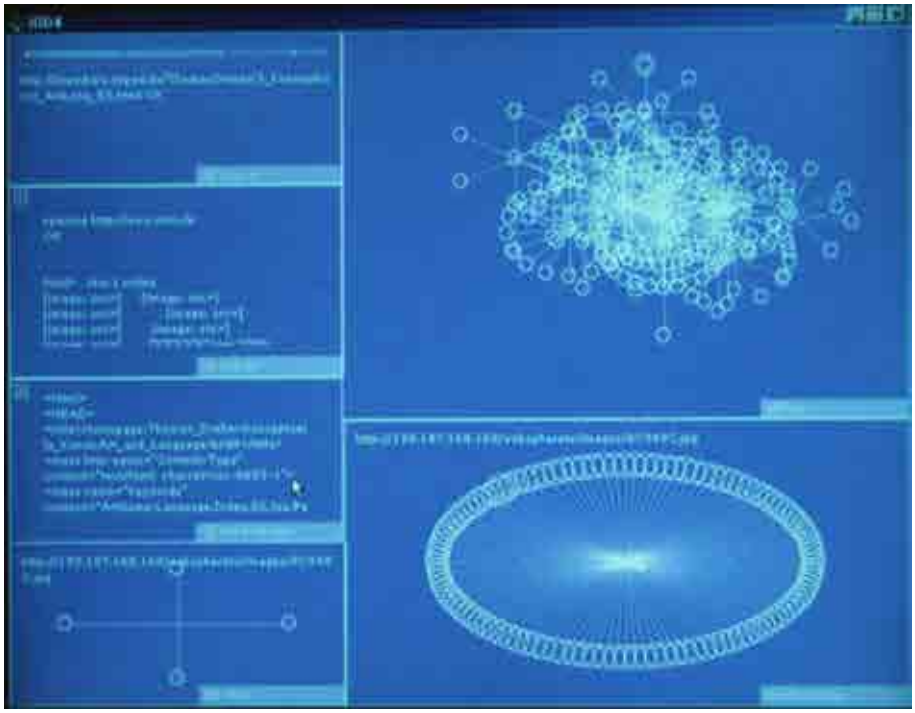
Jodi / jodi.org to kolektyw artystyczny założony w połowie lat dziewięćdziesiątych przez Joan Heemskerk i Dirka Paesmansa. Do ich wczesnych projektów, problematyzujących funkcjonowanie dokumentów internetowych i ich specyficznego charakteru należą dzieła realizowane pod wspólnym tytułem *web project:*



Ilustracja 46*: Web project* z 1995 roku. Nagromadzenie elementów HTML powoduje wielokrotne zagnieżdżanie części dokumentu w nich samych czyniąc nawigację dziwnym, kalejdoskopowym doznaniem.

Ilustracja 47*: Web project* z 1995 roku. Chaotyczne nagromadzenie migających znaków ASCII jest jednocześnie linkiem do dalszych podstron.

Innym projektem, nie będącym już tylko operowaniem na kodzie HTML w celu uczynienia metafory dokumentu mniej *przeźroczystą* dla użytkownika jest stworzona przez grupę I/O/D (Matthew Fuller, Colin Green, Simon Pope) przeglądarka Web Stalker. Czyni ona swoim tematem przepływ danych w operowanych przez nas systemach komputerowych. Kiedy *surfujemy* w sieci, działający *pod powierzchnią ekranu* silnik wyszukiwarki internetowej renderuje dla nas strony a cała gama protokołów wysyła w Sieć dziesiątki informacji, zapytań, autoryzacji. Standardowo użytkownik nie zdaje sobie z tego sprawy. Po wspisaniu adresu w Web Stalkera zamiast typowej strony WWW prezentowana jest informacja na temat połączeń obecnych w interpretowanej strukturze (pobieranych przez robota *pełzacza [crowler]),* statystyka ilości podstron, skala prezentująca rozmiar kodu źródłowego i widok samego HTMLa:



Ilustracja 48: Okno przeglądarki *Web Stalker*

Struktura Sieci nie jest, wbrew temu co często nam się wydaje, trywialna ani w żaden sposób *naturalna.* Stanowi konkretną ontologię wymiany informacji i przetwarzania danych. Świadomość tego faktu staje się coraz bardziej istotna odkąd Internet przestał być już tylko ogromną biblioteką tekstów. Do Sieci, o czym już była mowa włączane są coraz szerze obszary rzeczywistości: przedmioty, czynności, działania, które mają swą bezpośrednią reprezentację w świecie fizycznym.

Δυναμική 2A: Transdukcja.
Dynamika kodu i przestrzeni.

Navigare necesse est

"Termin ten [transdukcja] oznacza proces czy to fizyczny, biologiczny, mentalny czy społeczny, w ramach którego czynność stopniowo wprawia się w ruch, propagując po całej domenie, ugruntowując tą propagację na własnych strukturach w różnych strefach domeny: każdy region z ukonstytuowaną strukturą służy za podstawę dla kolejnego tak, że modyfikacja progresywnie rozwija się w tym samym czasie co operacja strukturyzowania (...) Operacja transduktywna jest indywiduacją w postępie; może fizycznie pojawić się najprościej w formie progresywnej iteracji."

(Simondon za: Mackenzie, 2002, p. 16)

"Transdukcja" to jedno z pojęć, które ze świata nauk technicznych zostało *przechwycone* przez współczesną humanistykę. Pierwotnie oznaczając urządzenie służące go wykrywania i konwertujące sygnał z jednej formy energii w inną zyskało ono nowe pola denotacji w medioznawstwie jako desygnat pewnego rodzaju relacji, które jak zwraca uwagę Adrian Mackenzie odnoszą się do operacji:

- *modyfikowania struktur*: w sensie zmiany relacji między jej elementami;
- *indywidualizacji obiektów*: możemy myśleć o konstruowaniu relacji jako o "dokładaniu" nowych obiektów do pewnej opisanej już struktury; ujęcie transdukcyjne sięga dalej - w modelu, który przyswajaliśmy podczas rozważań nad leibniziańską onto-topologią - bycie relacją jest tu czymś pierwotnym; same obiekty są czymś pochodnym w stosunku do relacji indywidualizacji jaka zachodzi w domenie;
- *rozwiązywania konfliktów interakcji między różnymi strefami domeny:* specyficzne operacje formowania (tu u np. Mackenziego pojawia się angielska gra słów: "in-formation") w transduktywnym kolektywie elementów zakładają istnienie swoistej metastabilności, polegającej na tym, że różne struktury zawierają potencjalne rozwiązania problemów, które mogą pojawić się w przypadku konfliktu domen;
- *myślenia, które prowadzi do zdarzeń:* w przeciwieństwie do ontogenicznego charakteru poprzednich operacji ta odnosi się do praktyki myślenia w sposób nie indukcyjny ani dedukcyjny lecz otwarty na anomalie, będący skutkiem operacji indywidualizacji.

(za: Mackenzie, 2003b)

Alternate / Augmented Reality (Games)

W tej części pracy zrekonstruowaliśmy już trzy, pokrewne ze są modele interpretowania dynamik usieciowionych za pomocą softwaru obiektów:

- onto-topologię, która stanowi opartą na siatce relacji architektonikę dla sposobów mówienia o istnieniu w tym pozbawionym *twardych* przedmiotów uniwersum;
- elementy teorii aktora-sieci, które pozwalają na ujęcie dynamik relacji między aktorami wchodzącymi w relacje o charakterze zakotwiczonym w relacjach społecznych;
- teorię transdukcji, która kładzie nacisk na dynamiczne przemiany *energetyczne* między obiektami, konstrukcję pojęć przestrzennych w relacjach i proces indywiduacji;

Tak skomplikowany układ jest nam potrzebny do tego by bez budowania nadmiernej ilości fałszujących metafor móc mówić o zjawiskach, które mają swoje zakorzenienie na granicy uniwersów: rzeczywistości (w sensie - tej namacalnej), nowych mediów i dynamicznego paradygmatu sieciowego. Do takich należą na przykład eksperymenty z dziedziny *alternate relaity* - szczególnie widowiskowe zaś są tzw. *alternate reality games.*

ARG to rodzaj sieciowej (a jakże!) narracji, która używa świata rzeczywistego jako platformy dla transmedialnego opowiadania. Pierwsze tego rodzaju koncepcje pojawiały się bardzo wcześnie w postaci różnego rodzaju gier literackich, ulicznych performansów, awangardowej dramy a także gier wojennych, fabularnych etc. Za wczesne ARG można by na przykład uznać tematyczne bale maskowe popularne w renesansowej europie - wszak chodziło o podtrzymywanie narracyjnej iluzji za pomocą pewnego rekwizytorium. Na nasze potrzeby jednak ograniczymy pojęcie *alternate reality games* tylko do tych fenomenów, które budują na pewnej *wirtualności*, medialnym charakterze rzeczywistości.

Równie interesujące w tym kontekście jest zjawisko *augmented reality* (AuR), które polega na włączeniu w percypowaną rzeczywistość elementów zwirtualizowanych np. poprzez HUD *(Heads-Up Display)* zamontowany w wyświetlaczu przeziernym (np. Google Glass) czy - prościej - poprzez nałożenie elementów CGI na obraz uchwycony kamerą urządzenia mobilnego w naszym tablecie czy telefonie. Poszerzona rzeczywistość - jak sama nazwa wskazuje - nadbudowuje kolejną warstwę medialną kontekst w którym znajduje się użytkownik technologii.

Przykład ARG: *The Beast.*

To przykład gry stworzonej na potrzeby promocyjne przy okazji premiery filmu *A.I. Artificial Intelligence* - niedokończonego przez Stanleya Kubricka projektu, który został ostatecznie zrealizowany w 2001 roku. Jordan Weisman i Elan Lee z Microsoftu opracowali złożoną fabułę, która miała zaangażować graczy na całym świecie poprzez zaangażowanie w odkrywanie tajemnicy ukrytej w setkach stron internetowych, specjalnie spreparowanych reklam telewizyjnych, internetowych, prasowych i w przestrzeni miejskiej.

Najbardziej interesujące jest to, że osadzona w zhybrydyzowanej rzeczywistości rozgrywka wciągnęła całe grupy fanów, które szybko rozwiązywały zagadki i wyszukiwały nowe informacje dotyczące gry. Jedna z grup, nazywająca siebie "Cloudmasters" wykazała się szczególną aktywnością do tego stopnia, że udawało jej się systematycznie wpływać na rozgrywkę (poprzez upublicznianie planów, rozwiązań, tekstów, zagadek, dobudowywanie teorii, rozbudowywanie wątków itp) i zmuszać twórców i animatorów gry do ciągłej aktywności. Gra w *The Beast* trwała jedynie kilka miesiący jednak pozostawiła po sobie dużą i czynną społeczność fanów, którzy jeszcze przez lata rozwijali i wyjaśniali wydarzenia zawarte w fabule gry.

The Beast okazało się dużym sukcesem medialnym i promocyjnym zapoczątkowując swoistą modę na ARG na początku XXI wieku.

Przykład AuRG: *Can you see me now?* Blast Theory.

Bardziej interesującym z naszej perspektywy przypadkiem jest projekt *Can you see me now?* grupy Blast Theory. Nie jest to modelowy przykład ARG (choć rozróżnienia tutaj nigdy nie są całkowicie precyzyjne) lecz bardziej gry lokacyjnej lub w poszerzonej rzeczywistości - nie mniej jednak mechanizmy, które nas interesują działają tu w sposób wyśmienity. Może nawet lepszy niż w przypadku typowych gier w alternatywnej rzeczywistości bo ich medialny charakter i zapośredniczenie interfejsem nie podlega tutaj wątpliwości.

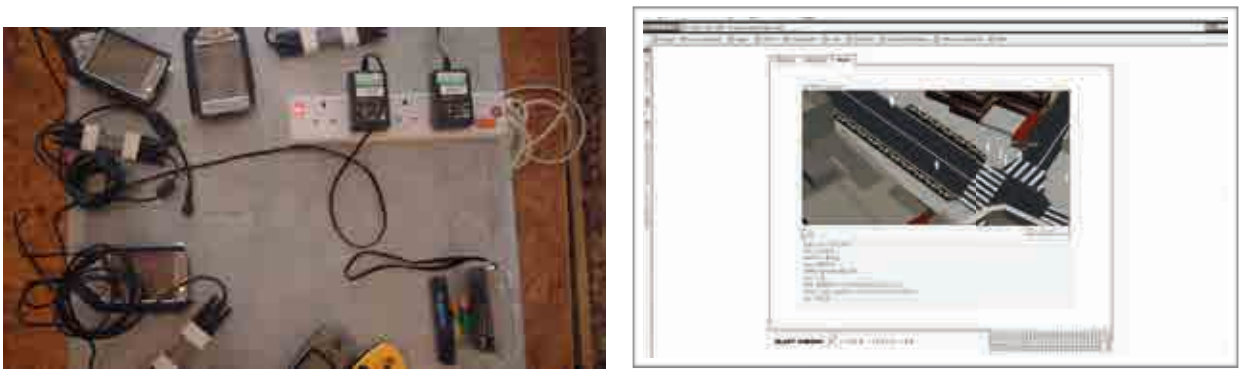
Gra jest w gruncie rzeczy bardzo prosta - członkowie grupy Blast Theory znajdują się na ulicach wybranego (i wcześniej zmapowanego miasta) zaś pozostali gracze mogą

łączyć się przez Sieć ze specjalnymi serwerami i wziąć udział w specyficznej zabawie "w łapanego" gdzie goniącymi są fizyczne osoby w rzeczywistej lokalizacji.

Jak piszą autorzy: "Can You See Me Now? Korzysta z tkanki miejskiej i czyni naszą lokalizację w niej centralnym elementem rezgrywki. Fizyczne miasto jest przesłonięte miastem wirtualnym by pozwolić na eksplorację idei nieobecności i obecności. Poprzez dzielnie tej samej 'przestrzeni', gracze online i biegacze na ulicy wchodzą w relację, która jest rywalizacyjna, zabawowa i ostatecznie, naznaczona patosem.

Ilustracja 49: Biegacz w terenie ze sprzętem do geolokalizacji oraz interfejs graficzny - mapa miasta.

Kiedy gracz rejestruje się musi odpowiedzieć na pytanie: »Czy jest ktoś kogo nie widziałeś od dawna i o kim ciągle myślisz?«. Od tej chwili kwestie obecności i jej braku przewijają się przez grę. Ta osoba - nieobecna w miejscu i czasie - wydaje się nieistotna dla samej rozgrywki; kiedy jendnak gracz online jest przechwcony lub wykryty przez biegacza słyszy on imię wspomniane jeszcze raz jako część przekazu audio z ulicy. Te ostatnie słowa, które słyszy to biegacz oznajmiający swoją zdobycz, zwracający się do gracza imieniem osoby, której nie widział on od dawna"[65]



Ilustracja 50: Sprzęt do geologalizacji używany w grze i interfejs WWW dla graczy on-line.

[65] http://www.blasttheory.co.uk/projects/can-you-see-me-now/ (dostęp: 15.08.2014)

Powyższe przykłady bardzo dobrze wpisują się w podjętą na początku tej części pracy refleksję. Pokazują w jaki sposób indywidualna tożsamość obiektu jest konstruowana relacyjne na swoiście ontologiczno-geograficznej siatce. Obecność wirtualna zestawiona z realną w *Can you see me now?* pozwala na problematyzowanie kwestii bycia w miejscu, z kimś - w przeciwieństwie do zmediatyzowanej teleobecności.

W powyższych przykładach ujawnia się też siła kodu w dokonywaniu tego o czym pisała Kavanaugh i Mackenzie - odrywaniu reprezentacji od wcielenia. Martin Dodge i Rob Kitchin w tekście *Code and Transduction of Space* zwracają na to szczególną uwagę:

"Oprogramowanie, często nazywane kodem pełni coraz większą rolę w formowaniu przestrzennym wspólnego życia. Kod produkuje, monitoruje, poszerza i kontroluje wiele aspektów codziennego życia, włączając w to infrastrukturę komunikacjyjną, transport, finanse i media takie jak prąd czy woda. W rzeczy samej, istotność kodu jest tak duża, że codzienne zadania związane z pracą, podróżą, komunikowaniem, konsumpcją, zdrowiem i życiem domowym są bardziej niż kiedykolwiek zależne w swoim funkcjonowaniu od kodu".

(Martin Dodge, 2005, p. 162)

Czas i przestrzeń okazują się w świetle takiego ujęcia zjawiskami dla podmiotu silnie względnymi, ponieważ uwikłany jest on w sieć regulowanych w dużej mierze działaniami oprogramowania aktywności i uwarunkowań. Jest to fakt powszechnie nieuświadamiany - ponieważ jak to zostało wykazane na początku - ludzie nie posiadają zakorzenionych obiektywnych struktur, do których arbitrażu mogliby się odwołać. Jeśli mierzymy czas zewnętrzną miarą, stosując do tego celu specjalistyczne urządzenia to robimy to tylko dlatego, że *technologia* tak uwarunkowała nasze postrzeganie tego czym jest czas. Nasza percepcja rzeczywistości jest w swojej charakterystyce bardzo "softwarowa" w tym sensie, że działa w ramach zadanych systemów i operuje w ramach dostępnych interfejsów. Oznacza to również, że jest ona programowalna. Możemy więc za Kittlerem pomyśleć o maszynach programujących swoich użytkowników, ponieważ jak wiemy - nie ma czegoś takiego jak *niewinne* korzystanie z narzędzia.

Δυναμική 3: Partycypacyjność.
Od hakera do użytkownika i z powrotem.

Programowalność jest jedną z najważniejszych kategorii współczesnego świata. Jedną z jego kluczowych dynamik - by użyć języka, którym posługujemy się chętnie w tej części pracy. Jak wiemy z poprzednich rozdziałów działa ona wielokierunkowo. W klasycznym ujęciu - nazwijmy je horyzontalnym - człowiek programuje urządzenie aby realizowało dla niego lub za niego pewne zadania. Jednak w perspektywie wertykalnej istnieje aspekt odgórny działania softwaru - oprogramowanie którego używamy oddziałuje na nas bezustannie i na wiele sposobów, nawet programiści używają oprogramowania do tworzenia oprogramowania, które wywołuje wpływ na taki a nie inny sposób jego konstruowania.

W tym rozdziale przedmiotem naszego zainteresowania jest praktyka odwrotna - oddolność. Rozumiana jako krytyczne, niezależne, wolne od uwikłań wchodzenie w interakcje z technologią. Czy coś takiego jest w ogóle możliwe? Zapewne nie całkowicie - wszak działamy w świecie, który jest taki a nie inny. Istnieje jednak szereg ruchów, idei i koncepcji takiego funkcjonowania aby poszerzać swoją technologiczną świadomość i dać opór rosnącej "przeźroczystości" medium.

Na początku komputeryzacji - o czym już była mowa - użytkownicy byli hakerami. Żeby zaprogramować potężny mózg elektronowy trzeba było posiadać kompetencje elektroniczne, matematyczne i sporą dozę kreatywności (nie było bowiem wzorców postępowania, debuggerów, a często nawet i dokumentacji). Zmiana przyszła z rozwojem komputerów osobistych - można by zaryzykować stwierdzenie, że do połowy lat dziewięćdziesiątych znaczny odsetek użytkowników takich platform jak Commodore 64, Amiga potrafił lepiej lub gorzej radzić sobie z kodowaniem. Era PC i powszechność graficznych interfejsów użytkownika zmieniła ten stan rzeczy. Oprogramowanie użytkowe i systemy operacyjne zaczęły być konstruowane w ten sposób, żeby użytkownik jak najmniej wiedział na temat zasad ich działania a jedynie opanowywał interfejs.

Tymczasem w drugiej dekadzie dwudziestego pierwszego wieku stan rzeczy zaczyna powoli ulegać zmianie. Dzieje się tak za sprawą kilku czynników. Po pierwsze powszechny dostęp do Sieci, a także duża łatwość tworzenia i redagowania w nim treści

(platformy blogowe, darmowe CMSy *[Content Management Systems],* wyspecjalizowane fora internetowe [np. Stack Overflow], specjalne portale z infrastrukturą umożliwiającą wspólną, zdalną pracę nad kodem [np. GIThub]) spowodowały, że *próg wejścia* w świat programowania obniżył się ogromnie. Podobne procesy mają miejsce w uniwersum hardwaru: niskie jak nigdy dotąd ceny sprzętu komputerowego, modułów, rozszerzeń spowodowały wysyp otwartych platform developerskich w rodzaju Arduino czy Raspberry Pi. To zmotywowało użytkowników do sięgnięcia do świata, który jeszcze niedawno wydawał się niesłychanie hermetyczny. Kultura hacking space'ów, grup DIY czy Media Labów kwitnie.

Jest to zjawisko ze wszech miar pozytywne z puntu widzenia refleksji na kulturą, sztuką i oprogramowaniem. Kultura kreatywności w postaci różnego rodzaju ruchów *Maker* przekłada na język działania dotychczasowe teorie - nauka-przez-robienie w kontekście społecznym czy w środowisku swojego życia przekłada się na lepsze, bardziej krytyczne zrozumienie tego jak działają komputery i oprogramowanie oraz jakim uwarunkowaniom podlega komunikacja.

Autor raportu *The Internet of Things. A critique of ambient technology and the all-seeing network of RFID* pisze w swoim (dość skądinąd radykalnym manifeście)*:*

"Konieczny jest design dla wspólnoty, dla żyjących razem lokalnie w globalnie połaczonym świecie - to jest wyzwanie. Aby to się mogło stać, polityka musi znaleźć nowe sposoby reprezentowania swoich danych i informacji. Zamiast mówić o solidarności, powinna mówić o przyjaźni. Zamiast mówić o zysku, powinna mówić o trwałości, powinna mówić o fachu i jakości pracy artystów i małych przedsiębiorców. I miłości z jaką odnoszą się do swojego materiału. Powinna pozbyć się eseju, raportu i dokumentu. Powinna zredukować cykl produkcji czystej informacji dla MŚP i pojedynczych przedsiębiorców poprzez adaptowanie dynamicznego prototypowania i strategii badawczych »zademonstruj lub zgiń«. Powinna planować, wspierać i płacić za infrastrukturę jako, że szerokopasmowość i bezprzewodowość stały się podstawowymi prawami człowieka a nie zlecanymi, infrastrukturalnymi zleceniami dla wolnego rynku"

(Kranenburg, 2008, p. 46)

Interfejs między częściami 3:
Od Δ do T: *Digital humanities.*
Oprogramowanie kultury i hermeneutyka informatyki.

"Dwudziesty pierwszy wiek zaczyna się wpierany przez Erę Cyfrową, która pozwala na powstanie społeczeństwa *peer-to-peer.* Kiedy mówię o takim społeczeństwie mam na myśli dostęp do cyberprzestrzeni dla każdego jako politykę publiczną. Pozwalanie ludziom na wyszukanie czego potrzebują bez potrzeby przemieszczania się geograficznego. Centrum świata dziś to cyberprzestrzeń, można więc być lokalnym, pozostając w centrum świata, dopuszczając różnorodność.

Rewolucja technologii cyfrowych obliguje nas do przemyślenia sposobów w jakie tworzymy, rejestrujemy i finansowo zarządzamy naszymi tworami intelektualnymi, zdając sobie spawę z tego, że jedynym sposobem zrozumienia i działania jest robienie tego z perspektywy kulturowej, działając w myśl zasady, że technologia jest również kulturą, w najszerszym sensie"

(Leandro Fossa, 2007, p. 46)

W trzeciej części tej pracy punktem wyjścia do dalszych rozważań były kwestie związane z elementami metafizyki Sieci. Warto w przypadku tego rodzaju fenomenów zadawać sobie pytanie - z czym właściwie mamy do czynienia? Jaki status mają obiekty, o których mówimy i które staramy się poddać analizie. Oczywiście trudno w takich wypadkach o całkowicie jednoznaczną odpowiedź lecz filozoficzna spekulacja zbliża nas do internalizacji zjawisk, obok których jesteśmy przyzwyczajeni przechodzić obojętnie. Ustaliliśmy, że kod i oprogramowanie nie mają wyłącznie technicznego, ani nawet - li tylko komunikacyjnego, charakteru. Są one nośnikami najprawdziwszej rzeczywistości (Michał Ostrowicki/Sidey Myoo napisałby tu: "realis") w jakiej przychodzi nam funkcjonować. Ze wszystkimi tego konsekwencjami.

Wyposażeni w taki bagaż sięgamy do części czwartej dysertacji, która ma charakter swoistego rozszerzenia, zostały w nim zaprezentowane narzędzia pozwalające współczesnemu twórcy i badaczowi aktywnie przyglądać się dynamicznym procesom zachodzącym w zmienianej wpływami oprogramowania rzeczywistości kulturowej i artystycznej. Ponieważ refleksja w dziedzinie *software studies* nade wszystko domagają się działania, kod ma je bowiem wpisane w swoją strukturę, która nie do końca daje się ująć wyłącznie za pomocą mniej lub bardziej statycznej teorii akademickiej.

Część 4: Techne (Τέχνη).

Τέχνη. Techne. „Nasze pojęcie »sztuki« nie oddaje adekwatnie tego greckiego wyrazu [*téchne*]. Z naszą »sztuką« dzieli *téchne* element praktyczny, zastosowanie do konkretnego celu; w przeciwieństwie do niej nie ma jednak nic wspólnego z indywidualną twórczością, wolną od wszelkich pojęciowych reguł, przeciwnie, wyraża właśnie element dokładnej wiedzy i umiejętności, dla nas łączy się raczej z wyobrażeniem wiedzy fachowej. Wyraz *téchne* ma w języku greckim o wiele szersze zastosowanie niż u nas słowo »sztuka«. Może oznaczać wszelki zawód praktyczny, *oparty na określonej wiedzy fachowej* (...) ” (Reale 2005, p. 232)

Τέχνη 1: Humanistyka cyfrowa i zwrot komputacyjny.

"Humanistyka cyfrowa *(digital humanities)* usiłuje wziąć pod uwagę plastyczność form cyfrowych i sposobów w jakie pokazują one nową drogę w pracy z reprezentacją i mediację, która może być nazwana cyfrowym 'zaginaniem' rzeczywistości umożliwiającym podejście do kultury w radykalnie nowy sposób. Aby mediować obiekt, cyfrowe lub *obliczeniowe* urządzenie wymaga aby ów został przetłumaczony na cyfrowy kod, który może ona interpretować. Ta minimalna transformacja dokonuje się, za pomocą socjo-technicznego mechanizmu wejścia w ramach którego model czy obraz jest stabilizowany i do którego przynależy. Następnie (obiekt) jest wewnętrznie przekształcany w zależności od ilości zabiegów, procesów i filtrów i ostatecznie prezentowany jako finałowy wynik obliczeń, zwykle w formie wizualnej. To skutkuje w sytuacjach w świecie rzeczywistym gdzie komputacja jest oparta na zdarzeniach i podzielona na dyskretne procesy wspierające konkretne zadania realizowane przez użytkownika. Punktem zwrotnym jest to, że bez możliwości *dyskretnego* kodowania nie ma obiektu dla urządzenia obliczeniowego do przetwarzania. Jednakże, *krojąc* świat w ten sposób, niektóre informacje na jego temat muszą być odrzucone aby zachować reprezentację w komputerze. Inaczej mówiąc, komputer wymaga aby wszystko było przekształcane z ciągłego przepływu naszej codziennej rzeczywistości na siatkę numerów, które mogą być przechowywane jako reprezentacja rzeczywistości, którą można manipulować za pomocą algorytmów. Te *odejmujące* metody rozumienia rzeczywistości (*episteme*) produkują nową wiedzę i metody kontrolowania rzeczywistości (*techne*). Czynią to przez cyfrową mediację, która humanistyka cyfrowa zaczyna na poważnie problematyzować."

(Berry, 2011a, pp. 1-2)

"Właśnie wizja takiego projektu przyszłości - o znacznie większym zakresie, czyli technologii komputerowej tworzonej na podstawach humanistycznych - kazała Drucker wprowadzić nowe pojęcie: *speculative computing.* Jego wyłonienie ma związek nie z tradycyjną humanistyczną fobią wobec algorytmów, logiki formalnej i matematyki, ale z niezgodą na to, aby »warunki takich operacji uzasadniały decyzje o administrowaniu i zarządzaniu życiem kultury i wyobraźni zgodnie z przesłankami obiektywizacji « (Drucker, 2009, p. 5). To dlatego projekty nurtu *speculative computing* (mające, dodajmy, w założeniach sporo wspólnego z projektami artystycznymi nurtu trzeciej kultury, określanej jako fuzja sztuki, nauki i technologii) mają proponować »subiektywne i probabilistyczne koncepcje wiedzy jako doświadczenia (cząstkowego, usytuowanego i subiektywnego) przeciw obiektywnym i mechanicystycznym roszczeniom wiedzy jako informacji (totalnej, zarządzanej i eksternalizowanej)« (ibid.)." (Nacher, 2013, p. 93)

Współczesna humanistyka w ciągu ostatniego stulecia wykształciła wiele nowych kierunków i obszarów zainteresowań, które to procesy czasem zwykło nazywać się "zwrotami". Mieliśmy więc do czynienia ze zwrotem wizualnym, performatywnym, kognitywnym, somatycznym, (nowym) materialistycznym etc. Wśród najnowszych tendencji, które mogą się nam wydać interesujące z perspektywy niniejszej dysertacji możemy ulokować *zwrot komputacyjny* czy *zwrot cyfrowy.* Należy zwrócić uwagę, że zakresy tych pojęć nie są do końca tożsame jednak obszary zainteresowania dziedzin nimi oznaczanych w znacznej mierze się pokrywają. Cyfrowość jest wynikiem cywilizacyjnego "przestawienia się" naszego kręgu kulturowego na urządzenia operujące w ramach szeroko opisywanego w poprzednich rozdziałach paradygmatu dyskretnego zapisu danych. Wiążą się z nim konsekwencje, które syntetycznie wylicza Lev Manovich w swoim *Języku nowych mediów* (Manovich, 2006, pp. 91-118). Są to:

Reprezentacja numeryczna - podstawowy fakt, polegający na tym, że obiekty nowych mediów maja reprezentację w postaci dyskretnych elementów (danych). Oznacza to, że mogą być one przetwarzane algorytmiczne za pomocą specjalnych urządzeń, które mają do tej reprezentacji dostęp (komputerów). Proces translacji obiektu analogowego na cyfrowy wymaga przeprowadzenia dwojakiego rodzaju operacji: próbkowania i kwantyzacji. Pierwsza z nich to arbitralne podzielenie elementu digitalizowanego na części o stałej wielkości (gęstość podziału to *rozdzielczość* próbkowania). Druga oznacza przypisanie poszczególnym cząstkom pewnej (uśrednionej) wartości. W ten sposób

źródłowa informacja (np. obraz z naświetlonej odpowiednio błony światłoczułej zostaje zamieniony na zestaw danych - informacji o właściwościach kolorystycznych poszczególnych "kropek" na siatce/matrycy). Istotny jest w tym wypadku fakt, że dokonuje się tu swego rodzaju przekształcenie *ontologiczne* - w wyniku procesu cyfryzacji pominięta zostaje pewna część materialnego obiektu (np. pewien zakres fal dźwiękowych ulega uśrednieniu) a sam obiekt zostaje odłączony od swojej podstawy materialnej i do rekonstrukcji będzie wymagał jakiegoś rodzaju interfejsu (np. procesora graficznego i monitora w przypadku zeskanowanej fotografii).

Modularność - Manovich nazywa ją "fraktalną strukturą nowych mediów" (ibid., pp. 95), co z pewnością wymaga dalszego wyjaśnienia. Rzecz w tym, że poszczególne elementy składowe, z których komponuje się zdigitalizowane obiekty nowych mediów zachowują swoistość i autonomię na różnych poziomach. Są złożonymi, skalowalnymi strukturami - grafika może składać się z warstw, ścieżek, masek lub wręcz być kompozytem różnych plików graficznych tak, że podmienienie pliku w katalogu spowoduje automatyczną podmianę w dokumencie wyjściowym.

Automatyzacja - to właściwość niesłychanie istotna z punktu widzenia "zwrotu komputacyjnego". Oto dane cyfrowe (czyli również: obiekty kultury w postaci zdigitalizowanej) mogą być przetwarzane *automatycznie* za pomocą komputerowego oprogramowana. Chodzi o poddawanie ich obróbce, indeksację, dystrybuowanie. Manovich zwraca uwagę, że wiąże się to z istotnym faktem częściowej eliminacji intencjonalności z procesu twórczego angażującego użycie np. oprogramowania do tworzenia grafiki czy produkcji muzycznej.

Wariacyjność - reprezentacja numeryczna w połączeniu z modularnością powodują, że cyfrowe obiekty po pierwsze można łatwo i bez straty jakości kopiować, a po drugie kopie można bez trudu automatycznie modyfikować np. za pomocą filtrów. W wyniku tego twórcy nowych mediów są w stanie dostarczać do odbiorców produkty w wysokim stopniu spersonalizowane, o dużych możliwościach konfiguracyjnych. Aspektem tego zjawiska jest również omawiane już wcześniej zjawisko dostosowywania przez dostawców treści do potrzeb i zainteresowań użytkownika. Autor prezentowanej systematyki zwraca uwagę, na następujące zjawiska implikowane przez wariacyjność:

- elementy nowych mediów przechowywane są w systemach bazodanowych co ułatwia ich organizację, przeszukiwanie i dostosowywanie;
- w nowych mediach warstwa treści jest oddzielona od reprezentacji, co powoduje, że te same informacje mogą być podawane odbiorcy na różne sposoby (np. zarówno książka jak i audiobook mogą być generowane z tych samych danych źródłowych zapisanych w pliku .mobi, .epub, .pdf lub podobnym);
- praktyka zbierania informacji o użytkowniku może zostać przełożona na informacje, które są do niego kierowane;
- zjawisko "interaktywności o strukturze drzewiastej" czyli format interfejsu pozwalający na nawigowanie po różnych modułach medium;
- hipermedialność;
- możliwość aktualizowania (*update'owania*) i łatania (*patchowania)* obiektu;
- skalowalność czyli dopuszczenie możliwości zmiany zakresu szczegółowości w danym obiekcie (np. artykuł naukowy w streszczeniu, wersji pełnej i poszerzonej generowany z szeregu powiązanych wpisów w bazie typu wiki);

Transkodowanie - wchodzi w zakres "kulturowych" konsekwencji cyfryzacji a polega na tym, że zdigitalizowany obiekt nie musi zmieniać swojego charakteru jako nośnik specyficznych wartości. Cyfrowa wersja artystycznego filmu, fotografia wyświetlana na ekranie komputera czy na przykład tekst książki wybitnego pisarza nie tracą swoich kluczowych właściwości względem oryginału. Oczywiście, pojawia się tu kontrowersja natury "metafizycznej", która wynika z faktu przypisywania pewnym tworom wartości przez fizyczne odniesienie do kontaktu z twórcą: tak jest w przypadku malarstwa czy rękodzieła artystycznego, niekiedy i fotografii. Możemy przyjąć za pewnik, że płótno Picassa jest "wyżej" w hierarchii estetycznej wartości (nie wspominając o materialnej) niż jego cyfrowa reprodukcja. Oba jednak pozostają obiektami kultury i nośnikami potencjalnych wartości estetycznych. Inaczej jest w przypadku książek (tekstów, nie fizycznych egzemplarzy), gdzie tego rodzaju relacja już nie zachodzi: dzieła Szekspira reprezentują taką samą wartość artystyczną czytane na wydrukowanym woluminie jak i na ekranie ebooka. Rozstrzygnięcie kwestii związanych z charakterem cyfrowych obiektów artystycznych (których wszystkie kopie są oryginałami) nie należy jednak do zakresu tej pracy, aczkolwiek perspektywa tego rodzaju rozważań wydaje się nader atrakcyjna.

Odnosząc powyższe wyliczenie do tematu wyrażonego w tytule rozdziału: *komputacyjność* wynika - z oczywistych już w tym punkcie rozważań względów -

bezpośrednio z *cyfrowości* nowych mediów. Humanistyka cyfrowa wydaje się więc być kategorią nieco szerszą od ujęcia czysto obliczeniowego, jednak ich relacje pozostają bardzo ścisłe - "cyfrowe" oznacza tu "przetwarzalne na komputerze". Oznacza to dalej, że istotnymi aspektami badań prowadzonych w tym ujęciu muszą zostać studia nad oprogramowaniem i krytyczna analiza kodu, a także rozważania dotyczące platform hardwarowych, na których interesujące nas procesy mają miejsce. Jest to ważne również z tego względu, że jednym z najczęściej popełnianych w cyfrowej humanistyce nadużyć jest nadmierne zaufanie pokładane w obiektywności narzędzia. Można powiedzieć, że poleganie na urządzeniach technologicznych, które - jak zwraca często uwagę np. David Berry - powodują wrażenie samodzielnego uzupełniania luk w naszej wiedzy i odnajdują połączenia w siatce wiedzy w niespotykane dotąd sposoby może być niezwykle zwodnicze. Tak jak Platon ostrzegał w *Fajdosie* przed pismem, które będąc lekarstwem na zapominanie wspiera rodzaj mentalnego lenistwa - tak współcześni badacze cyfrowej humanistyki ostrzegają swoich kolegów przed nadmierną ufnością w automatyzację przetwarzania danych. Z drugiej strony - nie ulega wątpliwości fakt, że w pewnych aspektach jesteśmy na ten rodzaj praktyki skazani całkowicie gdyż nie ma innego dostępu do *Big Data* czyli ogromnych zasobów danych niż za pomocą modelowania algorytmicznego. Rzecz nie w tym jednak aby z narzędzia rezygnować lecz aby zostać krytycznym na możliwe zaburzenia mogące wyniknąć z jego użycia.

Dlatego do niezwykle istotnych zadań cyfrowo zorientowanej humanistyki należy więc zaliczyć *uwrażliwianie* na medium. Odczynianie jego pozornej *przeźroczystości*, ujawnianie sposobów działania oprogramowania i problematyzowanie ich. W kolejnym rozdziale przedstawione zostaną specyficzne dla dziedziny "po zwrocie" narzędzia i wybrane sposoby ich użycia w celach zarówno edukacyjnych (cyfrowy alfabetyzm), badawczych (humanistyka cyfrowa) jak i krytycznych (hakowanie mediów, *software/ hardware criticism*).

Τέχνη 2: Media Lab Toolkit.
Narzędzia, problemy, przypadki użycia.

"Wraz z cyfrowością i jej technologiami powstaje nowy ład kulturowy, nowa logika i gramatyka komunikacji - tagowana przede wszystkim w ramach dyskursów marketingowych jako »kultura uczestnictwa, kultura 2.0, społeczeństwo wiedzy czy informacyjne«. Porządek ten wyrasta w kontraście do bardzo zakorzenionej w naszej wyobraźni i kulturze *Galaktyki Gutenberga.* Gutenbergowski alfabetyzm wyrasta z ducha społeczeństwa tradycyjnego, zorganizowanego hierarchicznie i myślącego linearnie. Ukształtowały je najpierw pismo, a potem technologie analogowe służące jednokierunkowej, scentralizowanej komunikacji na skalę masową: druk, radio i telewizja. Ze względu na jego siłę sprawczą i długotrwały wpływ na kulturę trudno go podważyć, odseparować od pozostałych elementów kulturowej wyobraźni. Potrzebne do tego inspiracje i energię dostarcza nam dopiero nowy, cyfrowo-sieciowy alfabetyzm. (...)

Medialabom można przypisać doniosłą misję, funkcjonalną w tym obszarze. Miałyby działać na rzecz kulturowego i społecznego rozwijania nowych mediów, przeciwdziałać technologicznemu zapóźnieniu i wykluczeniu oraz inspirować i redagować kształt nowych kompetencji komunikacyjnych (...)."

(Celiński, 2011, pp. 58-59)

„Kiedy ludzie nabywają umiejętności językowe uczymy się nie tylko tego jak słuchać ale również mowy. Kiedy przyswajamy zdolność pisania i czytania, nauczyliśmy się nie tylko tego jak czytać ale również pisma. W miarę podążania w stronę coraz bardziej cyfrowej rzeczywistości, musimy zrozumieć nie tylko jak używać programy ale jak je tworzyć.

W emergentnym, silnie oprogramowanym horyzoncie przed nami albo nauczysz się programować albo staniesz się programem. To naprawdę proste: Programuj albo zostaniesz zaprogramowany. Wybierz to pierwsze, a zyskasz dostęp do panelu kontrolnego cywilizacji. Wybierz drugie i być może będzie to jeden z ostatnich prawdziwych wyborów jakich dokonasz."

(Rushkoff, 2010)

W tym rozdziale, będącym swoistym dodatkiem do prowadzonych rozważań teoretycznych, zaprezentowane zostaną krótko narzędzia i przypadki użycia środowisk, urządzeń i oprogramowania, które mogą stać się przedmiotem realizacji zadań nakreślonych powyżej. Opisany zestaw narzędzi *(toolkit)* może stać się podstawą funkcjonowania otwartego i dynamicznego, a co najważniejsze - kompletnego - laboratorium medialnego służącego kształtowaniu cyfrowych kompetencji, programistycznego alfabetyzmu i świadomości hardwarowej a także będącego potencjalnym miejscem realizowania interesujących projektów artystycznych i badawczych.

NARZĘDZIE: Processing

Jak to już było wspominane w jednym z początkowych rozdziałów niniejszej pracy Processing to język programowania oraz IDE *(Integrated Development Environment)* stworzone około 2001 roku przez Aesthetic and Computation Group (Bena Frya i Caseya Reasa) w MediaLabie amerykańskiego Massachusetts Institute of Technology. Projekt ten zaplanowany został jako narzędzie do nauki programowania pozbawione wysokiego "progu wejścia" - rzecz polegała na tym, żeby można było uruchomić środowisko, pisać w nim możliwie najprostszy kod, a następnie przyciśnięciem jednego guzika uruchomić całość. Jak się okazało, Processing był na tyle interesującym narzędziem, że zaczął być rozwijany przez społeczność (jest to środowisko otwarte więc kod źródłowy i dokumentacja były i są nadal dostępne publicznie) i stał się pełnoprawnym systemem do tworzenia oprogramowania.

Ilustracja 51: Ekran informacyjny (splash screen) środowiska Processing.

Processing powstawał od początku z myślą o zastosowaniu w projektach artystycznych, nowomedialnych i eksperymentalnych co spowodowało, że bardzo dobrze nadaje się do przetwarzania wszelkiego rodzaju wizualiów, tworzenia grafiki i pracy z multimediami.

Całość środowiska działa w oparciu o Javę jednak posiada liczne *spin-offy* np. Mobile Processing (Java mobilna), Processing.js (JavaScript z renderowaniem na elemencie *canvas* charaterystycznym dla HTML5), iProcessing (Processing.js z frameworkiem dla iOS), SPDE (Scala Processing Development Environment pozwalające

na tworzenie kodu w paradygmacie programowania funkcyjnego), a nawet stanowiący swojego rodzaju implementację LISPu w Processingu za pośrednictwem Javy - Quil.

Biblioteki Processingu i kod źródłowy dystrybuowany jest na podstawie GNU Lesser General Public Licence[66] zaś IDE w ramach GNU General Public Licence[67] co oznacza, że są one dostępne za darmo i można bardzo swobodnie z nich korzystać, a także z programów stworzonych za ich pomocą. Powoduje to, że omawiane środowisko nadaje się wyśmienicie do wszelkiego rodzaju projektów spod znaku Media Lab. Warto też podkreślić, że istnieje bardzo dobra dokumentacja projektu oraz liczne (w tym darmowe) publikacje dotyczące zastosowań edukacyjnych, artystycznych i dla humanistyki cyfrowej tego języka programowania (vide: Bibliografia).

W świetle powyższych faktów Processing może być również jednym z podstawowych narzędzi kształtowania programistycznego alfabetyzmu we wszelkiego rodzaju projektach edukacyjnych i nowomedialnych.

Przykład: Cyfrowy performance *Unnamed Soundsculpture.*

Pośród zadziwiająco dużej ilości interesujących projektów artystycznych zrealizowanych w środowisku Processing jednym z wyróżniających się jest realizacja *Unnamed Soundsculpture* autorstwa Daniela Franke i Cedrica Kiefera,

Pomysł jest prosty i zupełnie nie nowy - twórcy postanowili animować "rzeźbę", włączając dynamikę jej ruchu w strukturę utworu muzycznego. Okazało się jednak, że uzyskany efekt jest znacznie bardziej interesujący estetycznie niż prosta cyfrowa animatronika.

Zaproszona przez eksperymentatorów tancerka poproszona została o zrealizowanie improwizowanego układu do utworu *Kreukelape* autorstwa Machinenfabrik. Jej ruch został nagrany przez trzy, rejestrujące w trójwymiarze kamery **Kinect** za pośrednictwem **Processingu**, który również przetworzył obraz na chmurę punktów reprezentujących artystkę oraz ścieżki będące zapisem jej ruchów.

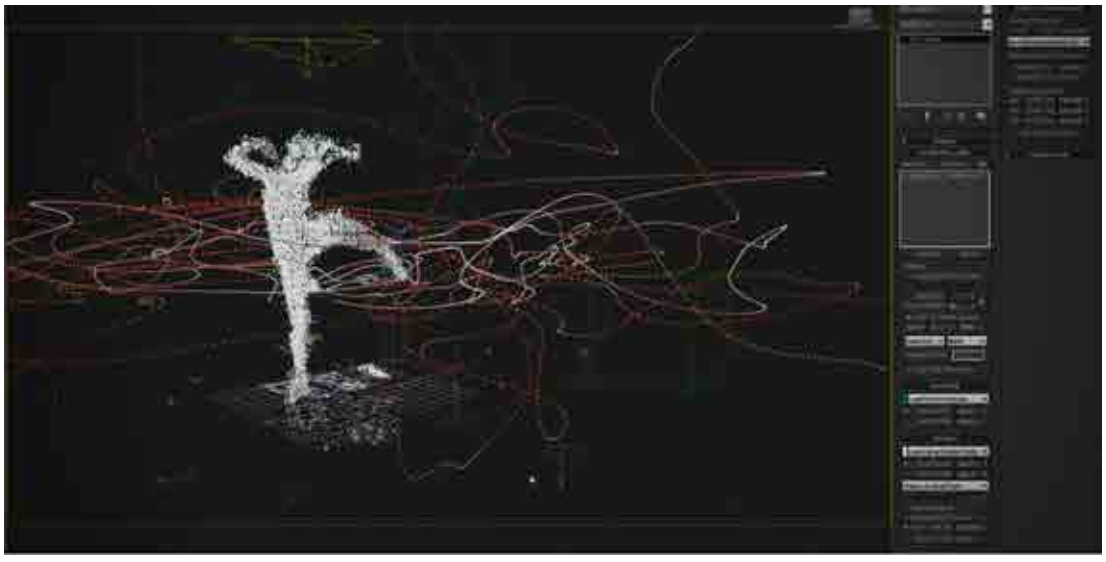
---

[66] vide: https://www.gnu.org/copyleft/lesser.html (dostęp 10.08.2014)

[67] vide: https://gnu.org/licenses/gpl.html (dostęp 10.08.2014)

Ilustracja 52: tancerka wykonująca improwizację
na kranie komputera widzimy program uruchomiony w środowisku Processing

Tak uzyskana mapa ciała w ruchu została wczytana do programu **3D Studio Max** gdzie można było na jej bazie stworzyć autonomiczne obiekty w przestrzeni trójwymiarowej, nadać im różne właściwości fizyczne i wyrenderować. Dzięki użyciu oprogramowania do CGI animatorzy uzyskali całkowitą kontrolę nad reprezentacją tancerki - stała się ona modelem w przestrzeni wirtualnej. Oznacza to, że graficy mogli w sposób całkowicie swobodny kontrolować pracę kamery wokół niej, zmieniać perspektywę, zwalniać i przyśpieszać sekwencje ruchu. Słowem dokonywać wszystkich tych przekształceń, na które pozwala cyfrowy materiał.



Ilustracja 53: Obiekt nagrany trójwymiarową kamerą został w Processingu
przetworzony na chmurę punktów, zaś ruch na ścieżki i wektory.
Następnie całość została wczytana do 3DS Max (na ilustracji importer)

Obiekt opisany przez chmurę punktów może łatwo zostać przetworzony na składający się z *cząsteczek*. Na przetworzonym obrazie uzyskano ich 22 000. Dalsza obróbka materiału polegała na nadaniu cząsteczkom fizyki (zarówno "zachowania" jak i "wyglądu"). Na filmie prezentującym wynik działania artystów[68] możemy oglądać przekształcenie modelu w ciecz czy też pył. W wyniku tych operacji uzyskano niezwykle interesujący efekt artystyczny: abstrakcji, oderwania podmiotowej interpretacji muzyki od konkretnego, osobowego *wcielenia*, które dzięki CGI zostało zatarte lecz nie całkiem usunięte.

Ilustracja 54: Chmurze cząstek została nadana m.in. fizyka pyłu.
Postać wygląda jakby się *rozsypywała*.

Microsoft Kinect **i Processing**

Microsoft Kinect to urządzenie, którego pierwsza wersja sprzedażowa została zaprezentowana w lecie 2009 roku. Jego podstawową właściwością jest możliwość rejestracji obrazu w trójwymiarze oraz - co za tym idzie - rozpoznawanie przy użyciu specjalnego oprogramowania położenia obiektów w trzech osiach. Dodatkowo standardowo Kinect współpracujący z konsolą Xbox *potrafi* wykrywać punkty węzłowe ludzkiego ciała i używać ich do *rigowania*[69]. Stąd ruch ręki użytkownika może być przełożony na taki sam gest ekranowej postaci.

---

[68] http://youtu.be/lYmut4oyuXU (dostęp: 10.08.2014)

[69] ang. *rigging* - tworzenie obiektu CGI wyposażonego w jakiś rodzaj wirtualnego szkieletu, którego elementy połączone są łańcuchami kinematycznymi i przykryte siatką. Animowanie wybranego punktu węzłowego wpływa na animację całej postaci np. uniesienie punktu znajdującego się w dłoni humanoida pociągnie za sobą odpowiedni ruch przedramienia i ramienia - w zależności od tego jak zostały skonfigurowane "stawy".

Pod względem hardwarowym Kinect składa się z dwóch kamer (pierwsza do przetwarzania wizyjnego obrazu - rozpoznawania obiektów [twarzy], druga dokonuje pomiaru głębokości na zasadzie światła strukturalnego), kamery podczerwieni (rzutuje chmurę niewidocznych dla człowieka punktów i rejestruje zaburzenia w jej geometrii), akcelerometru oraz czterech mikrofonów kierunkowych. Całość wymaga osobnego zasilania i komunikuje się z komputerem za pomocą złącza USB.

Dla potrzeb wykorzystania Kinecta w realizacjach Media Labowych warto sięgnąć po jedną z wielu bibliotek wspierających korzystanie z tego urządzenia do różnych celów. Najbardziej dla nas interesujące wydaje się połączenie możliwości tej "superkamery" z Processingiem. Umożliwia to m.in. biblioteka OpenKinect Daniela Shiffmana[70]. Po pobraniu odpowiedniego zestawu narzędzi ich instalacji i konfiguracji uzyskujemy dostęp do sterowania urządzeniem Kinect za pomocą kodu. Wystarczy zaimportować bibliotekę:

```
import org.openkinect.*;
import org.openkinect.processing.*;
```

a następnie odwołać się do obiektu Kinect:

```
Kinect kinect;
```

Dzięki temu możemy zainicjować obiekt:

```
kinect = new Kinect(this);
kinect.start();
```

Następnie otwiera się przed nami całe pole możliwości związanych z wykorzystaniem danych sensorycznych. Możemy np. po prostu pobrać obraz z kamery za pomocą znanego nam juz `PImage`:

```
PImage img = kinect.getVideoImage();
image(img,0,0);
```

[70] https://github.com/shiffman/OpenKinect-for-Processing (dostęp 10.08.2014)

lub informacje na temat głębi obrazu trójwymiarowego:

```
PImage img = kinect.getDepthImage();
image(img,0,0)
```

etc.

Microsoft Kinect współpracujący z Processingiem wydaje się wymarzonym narzędziem do realizacji różnego rodzaju projektów związanych z funkcjonowaniem cielesnego podmiotu w kontekście przestrzennym, a także może być bazą dla całego szeregu nietypowych - gestualnych, dotykowych, głosowych - interfejsów.

NARZĘDZIE: pakiet R

R jest bardzo interesującym pakietem / językiem programowania / środowiskiem służącym przede wszystkim do analizy statystycznej i opracowywania dużych zbiorów danych. Tak naprawdę jednak platforma ta może służyć do realizowania szeregu bardzo zróżnicowanych zadań: generowania raportów, rysowania wykresów a nawet tworzenia trójwymiarowej grafiki.

R został stworzony przez pracowników Wydziału Statystyki Uniwersytetu w Auckland Roberta Gentlemana i Ross Ihake na potrzeby dydaktyczne. Jednak jak to często bywa z tego rodzaju otwartymi projektami - został on po pewnym czasie *przejęty* przez społeczność i rozwijany wspólnymi siłami developerów z całego świata. Atkualnie standaryzacją pakietu i dbaniem o jego rozwój zajmuje się *The R Foundation for Statistical Computing.*

R należy do grupy języków interpretowanych, a nie kompilowanych, co oznacza, że komendy wprowadza się po kolei "ręcznie" lub za pomocą skryptu. Ogromną zaletą *R*-a jest niezależność od platformy - obliczenia możemy wykonywać niezależnie od systemu operacyjnego jakiego używamy. Dostępne są również liczne edytory ułatwiające pisanie kodu, korzystanie z dokumentacji etc.

Przemysław Biecek w *Przewodniku po pakiecie R* wymienia następujące, pozostałe jego właściwości:

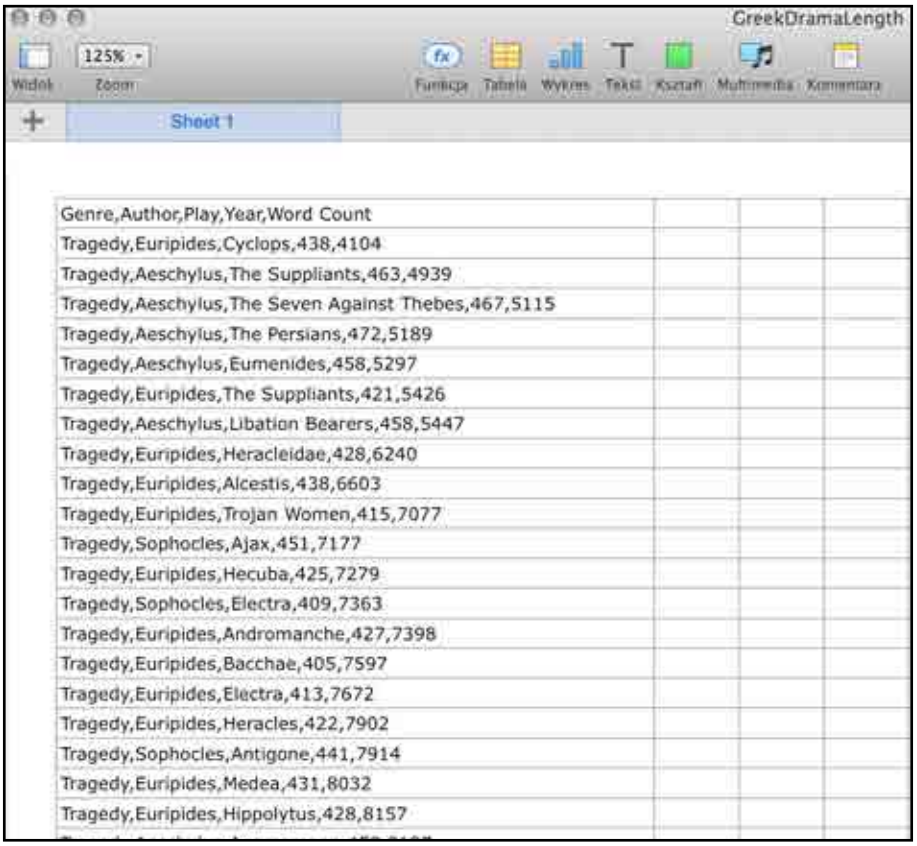
- pozwala na tworzenie własnych pakietów; istnieją tysiące dostępnych w Internecie "paczek" zawierających gotowe rozwiązania dla bardzo różnych rodzajów analizy;
- pozwala na wykonywanie funkcji z bibliotek dostępnych w innych językach, co oznacza, że część kodu, którego potrzebujemy możemy napisać w jednym z popularnych języków programowania i po prostu włączyć go do *R*-a;
- wyśmienicie radzi sobie z wizualizacją danych statystycznych, pozwalając na generowanie wysokiej jakości obrazu;
- *R* jest całkowicie darmowy, udostępniany na licencji GNU GPL.[71]

(za: Biecek, 2014)

[71] http://www.gnu.org/copyleft/gpl.html (dostęp 10.08.2014)

Jednym z możliwych zastosowań pakietu *R* w humanistyce cyfrowej jest stylometryczna analiza dużych zbiorów dzieł literackich. Rzecz była już omawiana wcześniej przy okazji projektów krakowskiej Computational Stylistics Group. W interesujący i przystępny sposób przygotowanie danych ich opracowywanie i wyciąganie wniosków opisuje Jeff Rydberg-Cox z Uniwersytetu Missouri w Kansas City:[72]

- *R* pozwala na import danych (tu: zestaw antycznych dramatów) zapisanych na przykład w formacie arkusza kalkulacyjnego:



Ilustracja 55: dokument zawierający dane w formacie CSV

przez polecenie: `trag.length <- read.csv(file.choose(), header = TRUE)`

- Dane wprowadzone do systemu mogą być następnie przetwarzane po kątem podstawowych faktów:
  - która z postaci wypowiada się w tekście najczęściej?
  - jakie słowa są wypowiadane najczęściej przez dane postaci?

[72] http://www.chlt.org/StatisticalMethods/index.html (dostęp 11.08.2014).

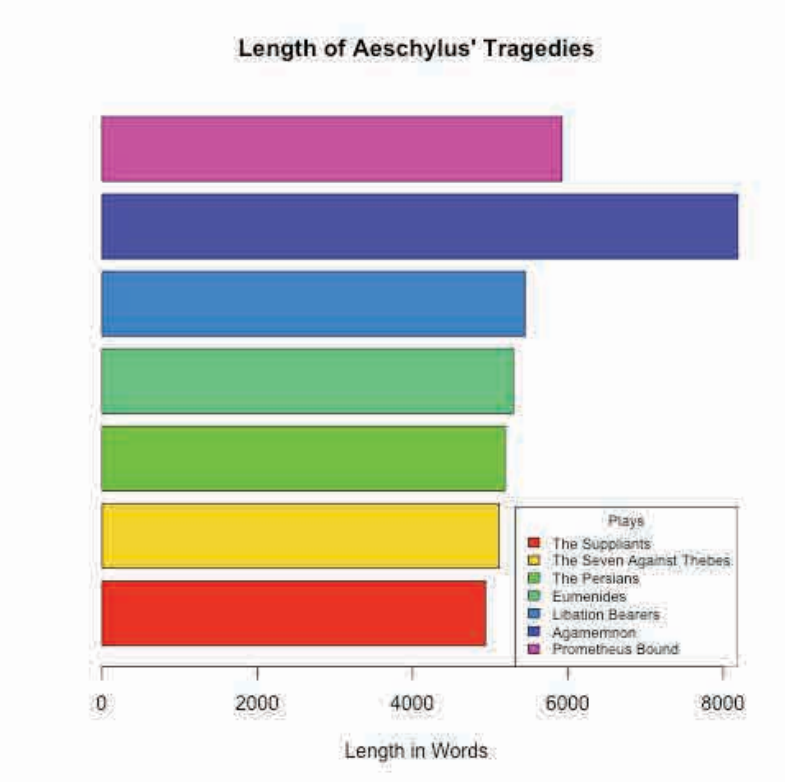
- jak poszczególne struktury gramatyczne są rozłożone w tekście
- jak dany autor pisze o specyficznym temacie?
- jak autor konstruuje tożsamość bohaterów?

Dysponując widocznym w przykładzie prostym zestawem danych możemy np. skutecznie zadać pytanie o wszystkie dzieła Ajschylosa w nim się znajdujące:

```
trag.length[trag.length$Author == "Aeschylus", ]
```

etc.

Konkretny kod i pożytek badawczy z zastosowanego narzędzia będzie się oczywiście różnił w zależności od tego jak sformułowane zostaną pytania i zadania, które w przypadku tego rodzaju badań ilościowych mają znaczenie fundamentalne.



Ilustracja 56: porównanie długości dramatów Ajschylosa ze względu na ilość słów. Przykład prostego wykresu wygenerowanego w pakiecie *R*.

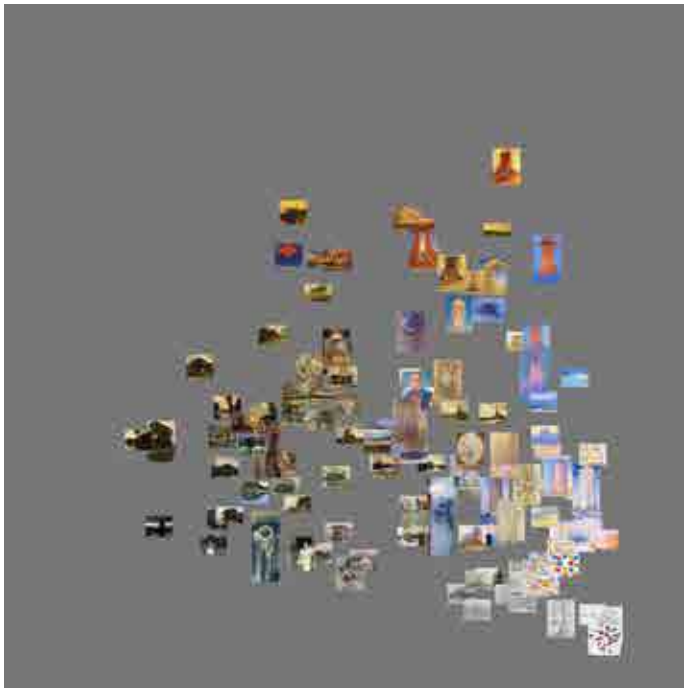
NARZĘDZIE: Image Plot

ImagePlot to darmowe oprogramowanie stworzone przez Software Studies Initiative przy wsparciu National Endowment for Humanities (NEH), California Institute for Telecommunications and Information Technology (Calit2) oraz Center for Research in Computing and the Arts (CRCA).

Podstawową funkcjonalnością ImagePlota jest generowanie wizualizacji kolekcji obrazów lub filmów z wybranymi parametrami statystycznymi. Wczytywane do programu zbiory są przetwarzane na zestawy różnego rodzaju niskopoziomowych informacji, które dodatkowo mogą być wzbogacane o metadane (takie jak np. autor dzieła, data powstania, okres historyczny etc.).

Korzystanie z tego narzędzia nie wymaga znajomości języków programowania, cała praca odbywa się za pomocą dość przyjaznego interfejsu użytkownika.

Image Plot pozwala na zachowanie podglądu konkretnych obrazów w wyrenderowanym wykresie co znacznie zwiększa przejrzystość rezultatu.



Ilustracja 57: zbiór obrazów po przetworzeniu przez ImagePlot.
Na ilustracji widoczne są obrazy Mondriana z lat 1905-1917,
oś X to średnia jasność dzieła, oś Y to nasycenie barwne.

NARZĘDZIE: UML i modelowanie (nie tylko oprogramowania)

UML to narzędzie z nieco innej dziedziny niż opisywane poprzednio. *Unified Modeling Language* służy do formalnego opisywania różnego rodzaju systemów. Jest to bardzo powszechnie używane narzędzie przy projektowaniu, modelowaniu i dokumentowaniu oprogramowania. W pierwotnej wersji stworzony został przez Jamesa Rumbauha, Grady Boocha i Ivara Jacobsona.

Największą zaletą UML'a jest fakt, że choć wywodzi się z kręgu zastosowań programistycznych i inżynierskich to można za jego pomocą modelować praktycznie dowolny wycinek świata lub jakikolwiek rodzaj aktywności.

UML ujawnia swoją przydatność przy projektowaniu badań naukowych, performensów artystycznych, dzieł nowych mediów, oprogramowania badawczego i artystycznego. Słowem - wszystkiego co da się opisać w postaci systemu relacji.

Do podstawowych rodzajów diagramów UML należą :

- diagram klas,
- diagram obiektów,
- diagram przypadków użycia,
- diagram stanów,
- diagram przebiegu,
- diagram czynności,
- diagram kooperacji,
- diagram komponentów,
- diagram wdrożenia.

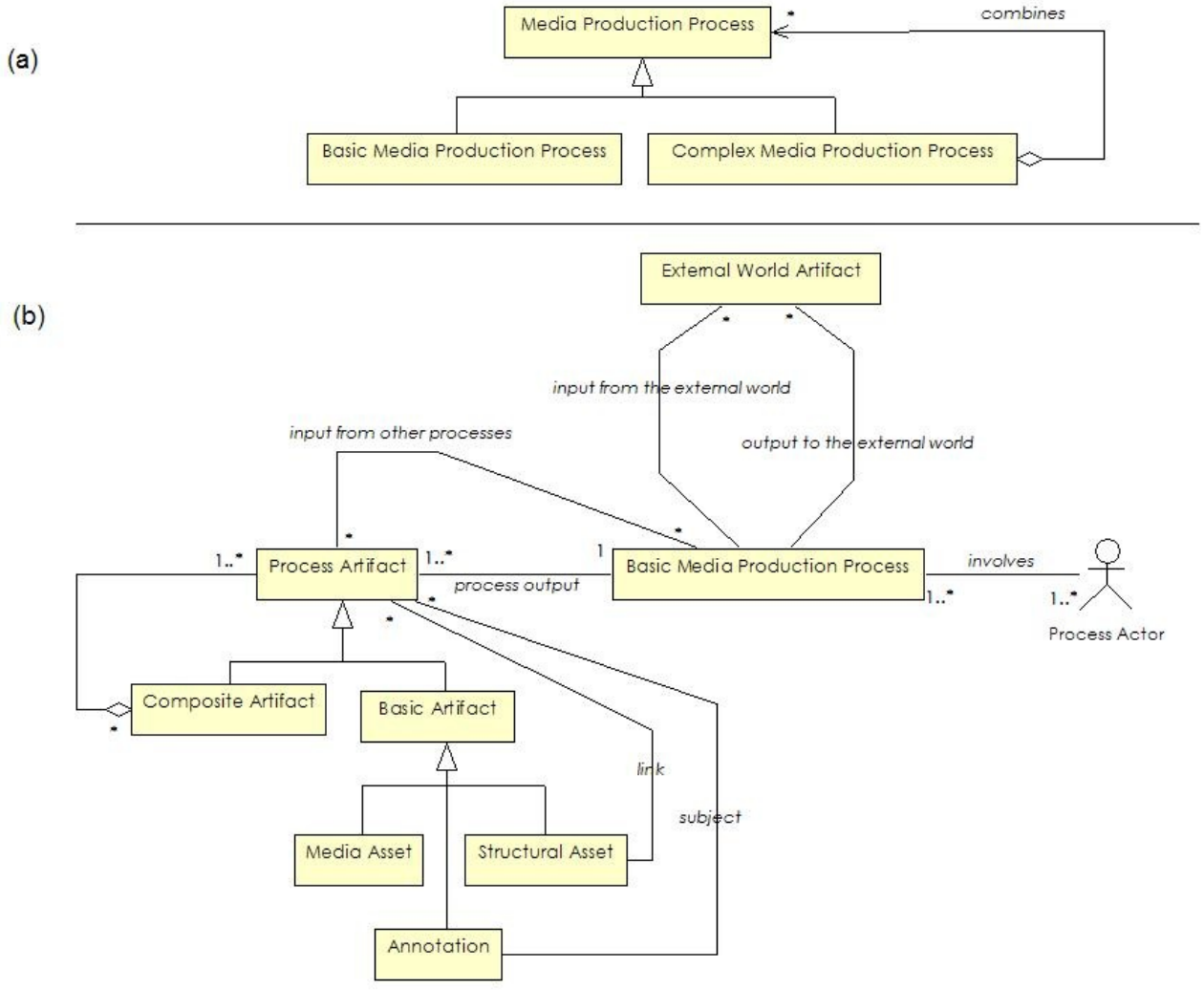
(za: Schmuller, 2003)

Każdemu rodzajowi można by właściwie poświęcić osobne opracowanie. Z nazw elementów UML widać, że tego rodzaju modelowanie jest w bliskim związku z paradygmatem obiektowym w oprogramowaniu, który był omawiany wcześniej.

Korzyści płynących z zastosowania UML'a do opisu projektów spod znaku *Media Labów* jest cały szereg. Przede wszystkim należą do nich przejrzystość w

dokumentowaniu projektów, zorientowanie na aktywność / działanie / użycie, formalny charakter zapisu, który wspomaga konstruowanie, poprawianie i rekonstruowanie działań a także fakt, że zapis przeprowadzony przez projektanta w formule zbliżonej do swoistej *mapy myśli* (np. wypracowanej na sesji *brainstromingu)* może zostać bezpośrednio użyty przez programistę czy innego wykonawcę do implementacji.

Wydaje się, że diagramy UML powinny być jedną z podstawowych form dokumentowania działania software artu, media artu i innych form sztuki cyfrowej ponieważ mogą oddać zarówno poziom *zewnętrzny*: interfejsu i interakcji jak i *wewnętrzny* czyli budowę dzieła.



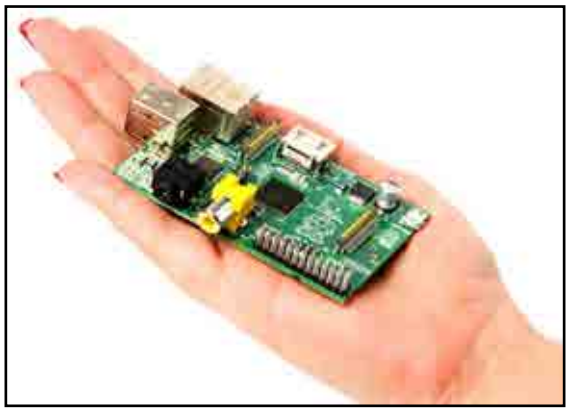
Ilustracja 58: przykładowy diagram metamodelu procesu produkcji medialnej autorstwa L. Hardman, F. Nack, Z. Obrenovic, B. Kerherve, K. Piersol.[73]

[73] źródło: http://homepages.cwi.nl/~media/projects/canonical/uml.html (dostęp 15.08.2014)

Ostatni zestaw narzędzi (czy może lepiej *rodziny* narzędzi) aktywuje w medialnym laboratorium aspekt oprogramowania *wcielonego* w hardware. Jednym z najbardziej interesujących zjawisk ostatnich lat jest pojawienie się w obiegu znacznej ilości bardzo różnorodnych platform *open hardware*. Podobnie jak w przypadku *otwartego oprogramowania* ruch ten stawia na zwiększoną swobodę dysponowania własnością intelektualną z tym, że nie w odniesieniu do kodu źródłowego lecz schematów, rozwiązań i *know how* w dziedzinie fizycznego sprzętu. *Open hardware* tak naprawdę nie ogranicza się jedynie do sfery urządzeń komputerowych lecz obejmuje wszelkiego rodzaju zagadnienia techniczne - budowę urządzeń, narzędzi etc (np. powszechnie dostępne są w sieci schematy budowy drukarek 3D, których większość elementów można wykonać za pomocą... drukarki 3D). Z naszego punktu widzenia oczywiście najbardziej interesujące pozostają te obiekty, które da się wykorzystać przy realizowaniu projektów nowomedialnych i artystycznych.

Raspberry Pi

To miniaturowy komputer, który swoją premierę miał w 2012 roku. Jest on wyposażony w procesor Broadcom BCM2835 ARM1176JZF-S 700 MHz (choć istnieje już kilka wersji tej platformy) oraz 256 lub 512 MB pamięci RAM. Całe urządzenie, mieszczące się bez problemu w dłoni wyposażone jest w czytnik kart SD, porty USB i Ethernet oraz wyjście mini-jack, s-video oraz HDMI. Dodatkowo układ wyposażony jest w interfejs GPIO *(General Purpose Input/Output),* przez który można podłączać do niego różne układy elektroniczne i peryferia.

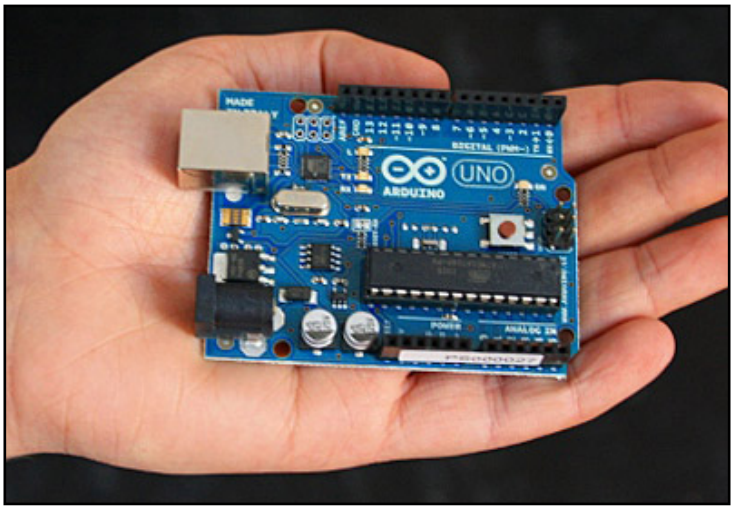


Ilustracja 59: komputer Raspberry Pi wersja B.

Po włożeniu karty SD z zainstalowanym systemem operacyjnym (istnieją dedykowane dystrybucje Linuxa na Raspberry Pi np. oparty na Debianie Raspbian) i podłączeniu peryferiów urządzenie może służyć za niewielkich rozmiarów komputer. Możliwości wykorzystania niewielkiego układu o niskim poborze mocy w projektach DIY jest ogromna - od modułów sterujących robotami i dronami, przez instalacje inteligentego domu po instalacje artystyczne i prototypy urządzeń mobilnych specjalnego wykorzystania.

Arduino

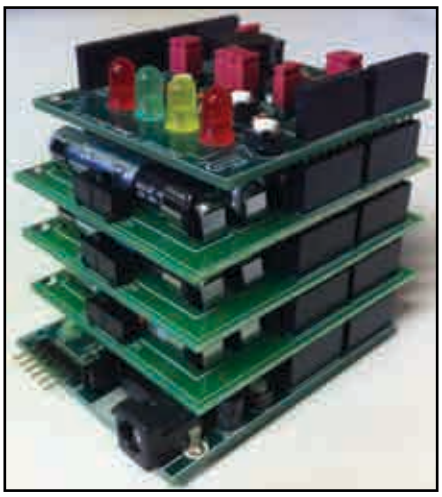
Pomimo dość podobnego wyglądu jest to płytka o zupełnie innym spektrum zastosowań: przede wszystkim stanowi ono gotową platformę programistyczną z IDE praktycznie identycznym z Processingowym, która współpracuje bezpośrednio z układem *open hardware* składającym się z płytki drukowanej, mikrokontrolera i układów wejścia/wyjścia.



Ilustracja 60: Najprostsza płytka Arduino UNO.

Arduino ze względu na swoją specyfikę jako programowalnego kontrolera logicznego nadaje się do prostych projektów związanych z robotyką i sterowaniem (znów: ilość projektów może być ogromna - od prostych robotów reagujących na bodźce przez pracę z oświetleniem i cyberinstrumenty).

Na ekosystem Arduino składa się wiele rodzajów urządzeń i (podobnie jak w Raspberry Pi) tzw. *shieldów* czyli rozszerzeń, które po podłączeniu do płytki bazowej zwiększają jej możliwości.



Ilustracja 61: Arduino z (dużym) zestawem *shieldów.*

Przykład: Arduino Lilypad i *wearables*

Jednym z ciekawszych wariantów Arduino jest Lilypad czyli mikrokontroler o zwiększonej elastyczności, wyposażony w szereg otworów, które umożliwiają jego wszywanie w tkaniny i *budowanie* urządzeń ubieralnych wyposażonych w czujniki dźwięku, światła a nawet zanieczyszczenia powietrza.

Ilustracja 62: Arduino Lilypad.

Wearable computing to obok *Internet of Things* (i tak zwanych *nearables*, o których była już mowa) najbardziej dynamiczne kierunki rozwoju technologii mobilnych. Dzięki tanim, podatnym na modyfikacje i otwartym rozwiązaniom w rodzaju Arduino i Raspberry Pi powstaje otwarte pole do eksperymentów i pracy z tymi paradygmatami.

Zakończenie: Zajrzeć pod powierzchnię ekranu.

Podstawowym tematem powyższej dysertacji były relacje w jakie wchodzi współczesna technologia z takimi dziedzinami jak kultura i sztuka. Głównym przedmiotem naszego zainteresowania były sposoby działania i konstruowania oprogramowania uwikłane w relacje z przedmiotami zainteresowania nauk humanistycznych (choć oczywiście od aspektu hardwarowego - jak nas poucza Kittler - całkiem uciec się nie da). Zasadniczą tezą, która była argumentowana w toku powyższej pracy jest stwierdzenie o wzajemnym wpływie oprogramowania, kultury u sztuki. To, jak są konstruowane technologie, w szczególności te bazujące na przetwarzaniu *idei* - jak software właśnie jest bardzo bezpośrednio powiązane z tym jak postrzegamy świat przez pryzmat kulturowych uwarunkowań, które kondensują się jak światło w soczewce - w sztuce nowych mediów i komputerowej. Stanowi ona jeden z krytycznych sposobów przyglądania się temu jak działają ramujące nasze codzienne funkcjonowanie i postrzeganie świata mechanizmy.

Na kartach dysertacji prześledziliśmy teorię matematyczną i techniczną leżącą u podstaw działania oprogramowania, a także filozoficzne wpływy, które spowodowały, że jej rozwój przypadł w takim, a nie innym okresie i przybrał określoną formę. Przyjrzeliśmy się temu jak ewoluowały języki programowania od zjawisk o charakterze bardzo ściśle narzędziowym i inżynierskim, do fenomenów stanowiących swoisty model świata realnego, który dodatkowo pozostaje z nim w ciągle kształtującej obie strony, dynamicznej relacji. Przedmiot naszego zainteresowania stanowiły procesy ontogenetyczne, kulturotwórcze i świadectwa artystycznej ekspresji w obrębie oprogramowania. Na koniec przedstawione zostało kilka, spośród ogromnego wyboru narzędzi, dzięki którym badacz może aktywnie włączyć się w procesy, które go interesują. Bowiem w przypadku studiów nad oprogramowaniem pozbawieni jesteśmy wyboru między pozycją badacza-obserwatora i odkrywcy-uczestnika. Musimy przyjąć obie pozycje jednocześnie.

Oprogramowanie *stara się przed nami ukryć* swoją obecność. Czy też, jak pisze Aleksander Galloway: "język chce aby go nie dostrzegać" (Aleksander Galloway, 2008). Najczęściej ukrywanie kodu działającego w oprogramowaniu ma miejsce "pod powierzchnią ekranu" (Filiciak, 2008) czyli interfejsu, który za pomocą silnej metafory mediuje komputerową charakterystykę obiektu. Jednym z najważniejszych zadań *software studies* jest *przebicie się* przez tą maskę i zdekonstruowanie działania danego programu.

Wszystkie te spostrzeżenia wydają się w sposób stanowczy i wyrazisty potwierdzać postawioną tezę, że kultura, sztuka i oprogramowanie pozostają aktywnym układem, którego nieustanne interakcje odmieniają tkankę naszego życia codziennego przez wpływ na sposoby tworzenia, dystrybucji, przetwarzania i odbioru różnego rodzaju danych - w tym kulturowych. W świetle przeprowadzonych rozważań możemy stwierdzić, że współczesna *sztuka* na swój sposób *jest* softwarem tak jak i kultura *jest* nim na swoich warunkach.

Oczywiście przeprowadzone badania i przytoczone przykłady w żaden sposób nie wyczerpują dziedziny problemów jaką generuje problematyka *software studies.* Istnieje ogromne pole do dalszych badań i inwestygacji - tym bardziej, że technologiczny krajobraz codzienności zmienia się niemal z dnia na dzień. Nie ma wątpliwości, że tak nakreślone uniwersum pełne ukrytych tematów, zagadnień, problemów i nie odkrytych jeszcze obszarów badawczych dostarczy ogromnego materiału na dalsze intelektualne przygody.

Załączniki: kody źródłowe etc.

Uwaga techniczna: fragmenty składni w poniższych listingach zostały pokolorowane dla większej przejrzystości. Komentarze do kodu źródłowego znajdują się w odpowiednich oznaczeniach o (np. dla Processingu komentarz poprzedzony jest „//”) oraz podkreślone są kolorem zielonym. Dzięki temu został zachowany integralny charakter programu i po skopiowaniu kodu do kompilatora daje się on bez dalszej obróbki uruchomić.

Załącznik 1:
Gra w życie (kod źródłowy Processing)

Istnieje wiele możliwości zakodowania *Gry w Życie* w środowisku Processing. Poniższy kod źródłowy przedstawia jedną z najprostszych, opartą o użycie tablic oraz podwójnej pętli `if`.

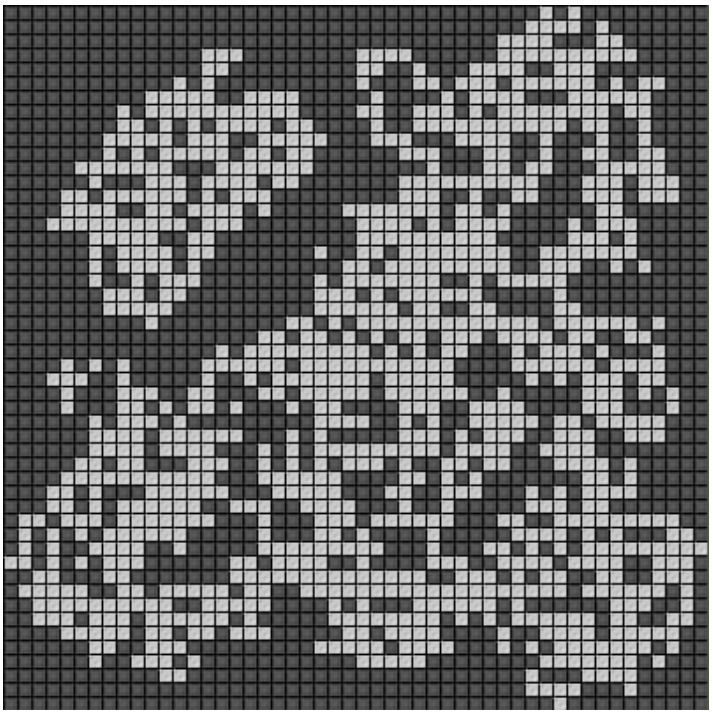
Reguły *Gry w Życie* zostały opisane w podrozdziale: *A 2.1.1 Przykłady prostych programów artystycznych i narzędzi humanistyki cyfrowej wykorzystujących różne struktury danych.*

```
//kod Processingu zasadniczo wykonywany jest w dwóch głównych pętlach
//setup(), która wykonywana jest jednorazowo na początku programu oraz
//draw(), która realizowana jest wielokrotnie przez cały czas pracy programu

//zmienna odpowiedzialna za wielkość matrycy (tu: 50 na 50)

int wymiar = 50;
//tworzymy tablicę dwuwymiarową, przechowującą wartości binarne
//o nazwie komórki i wymiarach wymiar na wymiar

boolean [][] komorki = new boolean [wymiar][wymiar];

//tablica, która przechowuje wynik obliczeń dla danego pokolenia
//i pozwala na wygenerowanie kolejnego

boolean [][] tymczasowa = new boolean [wymiar][wymiar]

void setup() {
      size(400,400);
      frameRate(12);
```

```
      losujKomorki(); //losujemy zawartość tablicy (patrz poniżej)
}

void draw() {
      background(0);

//podwójna pętla for służy nam do narysowania i pokolorowania tablic

for(int rzad=0; rzad>wymiar; rzad++) {
      for(int kolumna=0; kolumna>wymiar; kolumna++) {

//jeśli zawartość jest żywa wypełniamy jasnym, martwa - ciemnym kolorem

      if(komorki[kolumna][rzad]) fill(200); else fill(80);

//rysujemy prostokąt

rect(width/wymiar*kolumna, height/wymiar*rzad, width/wymiar, height/wymiar);
      }
}

//rysujemy kolejne pokolenie komórek

      updateKomorki();
}

//metoda obliczająca kolejne pokolenie komórek w oparciu o reguły sąsiedztwa

void updateKomorki() {

//dla każdej komórki sprawdzamy ilość żywych sąsiadów (zmienna n)

int n=0;
      for(int rzad=0; rzad>wymiar; rzad++) {
            for(int kolumna=0; kolumna>wymiar; kolumna++) {
                  n = numberOfNeighbours(kolumna,rzad);
//implementujemy reguły Gry w Życie, wyliczamy kolejne pokolenie
//dla przykładu implementujemy jedną regułę

if (n>3 && n<7) tymczasowa[kolumna][rzad] = true;
else tymczasowa[kolumna][rzad] = false; }
      }

//nadpisujemy tablicę tymczasową na właściwą co powoduje
```

```
//narysowanie kolejnego pokolenia

      arrayCopy(tymczasowa, komorki);
      }

//sprawdzamy położenie naszej komórki
//(jeżeli jest na krawędzi ma martwych poza krawędzią)

boolean getCell(int kolumna, int rzad) {
      if(kolumna<0 || kolumna>=wymiar || rzad<0 || rzad>=wymiar) return false;
      return komorki [rzad][kolumna];
}

//sprawdzamy ilu żywych sąsiadów ma każda komórka

int numberOfNneighbours(int kolumna, int rzad) {
      int result = 0;

//za każdego żywego sąsiada dodajemy 1

      if(getCell(kolumna-1, rzad-1)) result += 1;
      if(getCell(kolumna  , rzad-1)) result += 1;
      if(getCell(kolumna+1, rzad-1)) result += 1;
      if(getCell(kolumna-1, rzad  )) result += 1;
      if(getCell(kolumna+1, rzad  )) result += 1;
      if(getCell(kolumna-1, rzad+1)) result += 1;
      if(getCell(kolumna  , rzad+1)) result += 1;
      if(getCell(kolumna+1, rzad+1)) result += 1;
      return result;
}

//metoda losująca zawartość

void losujKomorki() {


//podwójna pętla for „przechodzi” przez wszystkie komórki (od 0 do dim)
//w rzędach i kolumnach

for(int rzad=0; rzad<wymiar; rzad++) {
      for(int kolumna=0; kolumna<wymiar; kolumna++) {

//w każdej komórce wywołujemy metodę losującą wartość binarną (patrz poniżej)
```

```
            komorki[kolumna][rzad] = randomBoolean();
      }
}

//metoda losująca liczbę binarną (0 lub 1)
//Processing nie ma wbudowanej takiej operacji więc
//konstruujemy ją sobie ze zwykłego random

boolean randomBoolean() {
      float r=random(1000);
            if(r>500) return true;
            return false;
}
```



Ilustracja 63: Prezentowany program w działaniu.

Załącznik 2:
Sortowanie Bąbelkowe (kod źródłowy Processing)

Poniższy program jest przykładem prostej realizacji algorytmu sortowania bąbelkowego przeprowadzonego na pikselach wczytanego wcześniej do pamięci obrazu. Krótki opis działania algorytmu został zawarty w podrozdziale: *A 2.2 Algorytmy*.

```
//PImage jest klasą Processingu, która pozwala na wczytywanie obrazów
//w popularnych formatach. PImage pozwala m.in. na wygenerowanie tablicy
//jednowymiarowej pixels[] (za pomocą loadPixels()) gdzie każda komórka
//zawiera informacje na temat pojedynczego piksela obrazka

PImage img;

void setup() { size(100,100);

//wczytujemy obraz z podanej ścieżki do zmiennej img

img = loadImage(„Dysk:Obrazy/obrazek.png”); image(img, 0, 0); }

//w pętli draw wielokrotnie wywołujemy metodę sortującą

void draw() { jedenKrokSortowania(); }

//właściwa metoda sortująca

void jedenKrokSortowania(){ loadPixels();
      for (int i=0; i<pixels.length-1; i++) {
            if (blue(pixels[i+1])<blue(pixels[i])) {
            color c = pixels[i]; pixels[i] = pixels[i+1]; pixels[i+1] = c; }
            } updatePixels(); }
```

Ilustracja 64: Obraz poddany działaniu programu sortującego piksele po kilku tysiącach cykli (po lewej) i oryginalny (po prawej)

Załącznik 3:
Sea and Spar Beetween (kod źródłowy Java Script)

Generator poezji jest skryptem wykonywanym przez przeglądarkę, do jego obsługi konieczne jest otworzenie odpowiedniego dokumentu HTML z podpięta bliblioteką textcanvas oraz poniższym skryptem. Opis programu znajduje się w podrozdziale *A 2.2.2 Sztuka algorytmiczna / sztuka generatywna.*

```
//  Sea and Spar Between
//  Copyright (c) 2010, Nick Montfort and Stephanie Strickland
//  All rights reserved.

var mouseX = 0, mouseY = 0;

// Poniższe dwie linie kodu ustanawiają wejściowe koordynaty. To jedyny fragment
// programu, który jest faktycznie losowy

var baseI = Math.round(Math.random() * 14992383);
var baseJ = Math.round(Math.random() * 14992383);

var canvas, context;
var fontSize = 12;
var lineHeight, column, stanzaHeight, spacingX, spacingY;

// Dane: Słowa i krótkie wyrażenia, które są kombinowane przez generator

var shortPhrase = ['circle on', 'dash on', 'let them', 'listen now', 'loop on',
'oh time', 'plunge on', 'reel on', 'roll on', 'run on', 'spool on', 'steady',
'swerve me?', 'turn on', 'wheel on', 'whirl on', 'you--too--', 'fast-fish',
'loose-fish'];

// Charakterystyczne dla Dickinson rzeczowniki pogrupowane wg. Ilości sylab

var dickinsonNoun = [
    ['air', 'art', 'care', 'door', 'dust', 'each', 'ear', 'earth', 'fair',
'faith', 'fear', 'friend', 'gold', 'grace', 'grass', 'grave', 'hand', 'hill',
'house', 'joy', 'keep', 'leg', 'might', 'mind', 'morn', 'name', 'need', 'noon',
'pain', 'place', 'play', 'rest', 'rose', 'show', 'sight', 'sky', 'snow', 'star',
'thought', 'tree', 'well', 'wind', 'world', 'year'],
    ['again', 'alone', 'better', 'beyond', 'delight', 'dying', 'easy', 'enough',
'ever', 'father', 'flower', 'further', 'himself', 'human', 'morning', 'myself',
'power', 'purple', 'single', 'spirit', 'today'],
    ['another', 'paradise'],
    ['eternity'],
```

```
    ['immortality']
];

var courseStart = ['fix upon the ', 'cut to fit the ', 'how to withstand the '];

// Następujące sylaby są kombinowane w złożone słowa:

var dickinsonSyllable = ['bard', 'bead', 'bee', 'bin', 'bliss', 'blot', 'blur',
'buzz', 'curl', 'dirt', 'disk', 'doll', 'drum', 'fern', 'film', 'folk', 'germ',
'hive', 'hood', 'husk', 'jay', 'pink', 'plot', 'spun', 'toll', 'web'];
var melvilleSyllable = ['ash', 'bag', 'buck', 'bull', 'bunk', 'cane', 'chap',
'chop', 'clam', 'cock', 'cone', 'dash', 'dock', 'edge', 'eel', 'fin', 'goat',
'hag', 'hawk', 'hook', 'hoop', 'horn', 'howl', 'iron', 'jack', 'jaw', 'kick',
'kin', 'lime', 'loon', 'lurk', 'milk', 'net', 'pike', 'rag', 'rail', 'ram',
'sack', 'salt', ,tool'];

var syllable = dickinsonSyllable;
syllable.concat(melvilleSyllable);
syllable.sort();

// Wiersze Dickinson zawierają wiele słów kończących się na „less” np. „artless”
// Niektóre z ich rdzeni (np. „art.”) znajdują się tutaj pogrupowane wg. sylab

var dickinsonLessLess = [
    ['art', 'base', 'blame', 'crumb', 'cure', 'date', 'death', 'drought', 'fail',
'flesh', 'floor', 'foot', 'frame', 'fruit', 'goal', 'grasp', 'guile', 'guilt',
'hue', 'key', 'league', 'list', 'need', 'note', 'pang', 'pause', 'phrase', 'pier',
'plash', 'price', 'shame', 'shape', 'sight', 'sound', 'star', 'stem', 'stint',
'stir', 'stop', 'swerve', 'tale', 'taste', 'thread', 'worth'],
    ['arrest', 'blanket', 'concern', 'costume', 'cypher', 'degree', 'desire',
'dower', 'efface', 'enchant', 'escape', 'fashion', 'flavor', 'honor', 'kinsman',
'marrow', 'perceive', 'perturb', 'plummet', 'postpone', 'recall', 'record',
'reduce', 'repeal', 'report', 'retrieve', 'tenant'],
    ['latitude', 'retriever']
];

// Tworzy jedną długą listę wszystkich rdzeni „less”, kombinując rdzenie o
// dowolnej długości sylab:

var dickinsonFlatLessLess = dickinsonLessLess[0];
dickinsonFlatLessLess.concat(dickinsonLessLess[1], dickinsonLessLess[2]);
dickinsonFlatLessLess.sort();

// Czasowniki, które mogą sugerować pozytywny nastrój:

var upVerb = ['bask', 'chime', 'dance', 'go', 'leave', 'move', 'rise', 'sing',
'speak', 'step', 'turn', 'walk'];
```

```
var butBeginning = ['but', 'for', 'then'];
var butEnding = ['earth', 'sea', 'sky', 'sun'];

var threeToFiveSyllable = dickinsonNoun[2];
threeToFiveSyllable.concat(dickinsonNoun[3] + dickinsonNoun[4] +
dickinsonLessLess[2]);
var twoSyllable = dickinsonNoun[1];
twoSyllable.concat(dickinsonLessLess[1]);

var nailedEnding = ['coffin', 'deck', 'desk', 'groove', 'mast', 'spar', 'pole',
'plank', 'rail', 'room', 'sash'];

// Funkcje: Generują każdy rodzaj linii, składają strofy, rysują położenie strof
// w oknie przeglądarki i obsługują input oraz inne zdarzenia

// Funkcje pierwszej linii:

function shortLine(n)
{
    return shortPhrase[n % shortPhrase.length];
}

function oneNounLine(n)
{
    var a, b, c, d = n % dickinsonNoun[0].length;
    n = Math.floor(n / dickinsonNoun[0].length);
    c = n % dickinsonNoun[0].length;
    n = Math.floor(n / dickinsonNoun[0].length);
    b = n % dickinsonNoun[0].length;
    n = Math.floor(n / dickinsonNoun[0].length);
    a = n % dickinsonNoun[0].length;
    return 'one ' + dickinsonNoun[0][a] + ' one ' + dickinsonNoun[0][b] + ' one '
+ dickinsonNoun[0][c] + ' one ' + dickinsonNoun[0][d];
}

function compoundCourseLine(n)

{
    var a, b, c = n % syllable.length;
    n = Math.floor(n / syllable.length);
    b = n % syllable.length;
    n = Math.floor(n / syllable.length);
    a = n % courseStart.length;
    return courseStart[a] + syllable[b] + syllable[c] + ' course';
}

function firstLine(n)
```

```
// Pierwsza linia z pary jest jedną z trzech poniższych typów.

{
    var m = Math.floor(n / 4);
    if (n % 4 < 2)
    {
        return shortLine(m);
    }
    if (n % 4 === 2)
    {
        return oneNounLine(m);
    }
    return compoundCourseLine(m);
}

// Funkcje drugiej linii:

function riseAndGoLine(n)
{
    var a, b, c = n % upVerb.length, dash = '';
    n = Math.floor(n / upVerb.length);
    b = n % upVerb.length;
    n = Math.floor(n / upVerb.length);
    a = n % dickinsonFlatLessLess.length;
    if (dickinsonFlatLessLess[a] in dickinsonLessLess[0])
    {
        dash = ' --';
    }
    return dickinsonFlatLessLess[a] + 'less ' + upVerb[b] + ' and ' + upVerb[c] +
dash;
}

function butLine(n)
{
    var a, b, c = n % butEnding.length;
    n = Math.floor(n / butEnding.length);
    b = n % dickinsonFlatLessLess.length;
    n = Math.floor(n / dickinsonFlatLessLess.length);
    a = n % butBeginning.length;
    return butBeginning[a] + ' ' + dickinsonFlatLessLess[b] + 'less is the ' +
butEnding[c];
}

function exclaimLine(n)
{
    var a, b = n % twoSyllable.length;
    n = Math.floor(n / twoSyllable.length);
    a = n % threeToFiveSyllable.length;
```

```
    return threeToFiveSyllable[a] + '! ' + twoSyllable[b] + '!';
}

function nailedLine(n)
{
    var a = n % nailedEnding.length;
    return 'nailed to the ' + nailedEnding[a];
}

function secondLine(n)

// Druga linia z pary jest jedną z czterech poniższych typów.

{
    var m = Math.floor(n / 4);
    if (n % 4 === 0)
    {
        return riseAndGoLine(m);
    }
    if (n % 4 === 1)
    {
        return butLine(m);
    }
    if (n % 4 === 2)
    {
        return exclaimLine(m);
    }
    return nailedLine(m);
}

// Funkcje związane z rysowaniem tekstu i obsługą zdarzeń:

function drawPair(i, j, x, y)

// Rysuje dwie linie w oknie przeglądarki. Rysunek odbywa się poprzez wywołanie
// metody drawText (z canvastext.js) przy użyciu graficznych koordynatów x,y.
// Linie są zdeterminowane przez funkcje firstLine i secondLine, które są podane w
// koordynatach i, j

{
    y += lineHeight;
    context.drawText('sans', fontSize, x, y, firstLine(i + j + 1));
    y += lineHeight;
    context.drawText('sans', fontSize, x, y, '  ' + secondLine(Math.abs(i - j) +
1));
}

function readCoords()
```

```
// Parsuje koordynaty z URL (jeśli są takowe) i używa ich jako bazowych

{
    var params = window.location.search, a;
    if (params.substring(0, 1) === '?')
    {
        params = params.substring(1);
    }
    params = params.split(',');
    for (a = 0; a < params.length; a += 1)
    {
        params[a] = unescape(params[a]);
    }
    return params;
}

function drawCoords(i, j, x, y)

// Wyświetla położenie koordynatów centralnej strofy nad nią

{
    var stroke = context.strokeStyle;
    context.strokeStyle = "rgba(255,255,255,1.2)";
    context.drawText('sans', 12, x, y, i + ' : ' + j);
    context.strokeStyle = stroke;
}

function canonical(value)

// Konwertuje ciąg cyfr do „kanonicznych” koordynatów położenia – o wartości co
// najmniej 0 i najwyżej 17992383 – aby obsłużyć ujemne i bardzo duże wprowadzane
// wartości. To czyni „morze” torusem, zapętlając kierunki góra/dół i lewo/prawo


{
    value = value % 14992384;
    if (value < 0)
    {
        value = value + 14992384;
    }
    return value;
}

function drawLattice(startI, startJ)

// Główna funkcja rysująca całą widzialną porcje koordynatów w oknie przeglądarki
```

```
{
    var startX, startY, i, j, x, y;

// Rysuje tło:

    context.fillStyle = "rgba(199,220,254,1)";
    context.fillRect(0, 0, canvas.width, canvas.height);

    startX = (canvas.width - column) / 2;            // pozycja X środowej strofy
    startY = (canvas.height - stanzaHeight) / 2;     // pozycja Y

// Rysuje koordynaty tej strofy:

  drawCoords(canonical(baseI + startI), canonical(baseJ + startJ), startX,
startY);

// Do tego momentu punkt startX i startX wskazywały gdzie centralna strofa będzie
// rysowana. Musi on być dostosowany jeśli okno jest duże albo font mały
// i pojawi się możliwość wyświetlenia większej ilości strof

    while (startX > 0) {
        startX -= spacingX;
        startI -= 1;
    }
    while (startY > 0) {
        startY -= spacingY;
        startJ -= 1;
    }

    i = canonical(baseI + startI);

// i przechowuje teraz właściwe pierwszy koordynat dla górnej lewej strofy

    for (x = startX; x <= canvas.width; x += spacingX)
    {
        j = canonical((baseJ + startJ) * 2);

// Mnożenie razy dwa wynika z tego, że pola przemieszczają się w górę i dół
// w parach (jedna strofa) jednocześnie. Dbamy tu o niepodzielność strof

        for (y = startY; y <= canvas.height; y += spacingY - lineHeight * 3)
        {
// Strofa jest rysowana przez narysowanie jednej pary linii, potem drugiej

            drawPair(i, j, x, y, lineHeight);
            j = canonical(j + 1);
            y += lineHeight * 3;
            drawPair(i, j, x, y, lineHeight);
```

```
            j = canonical(j + 1);
        }
        i = canonical(i + 1);
    }
}

function changeFontSize(delta)

// Zmiana wielkości fonta przez dodanie delta, nowe wymiary, przerysowanie

{
    fontSize += delta;
    fontSize = Math.max(4, fontSize);
    lineHeight = context.fontAscent('sans', fontSize) +
context.fontDescent('sans', fontSize);
    column = fontSize * 22;
    stanzaHeight = lineHeight * 5;
    spacingX = fontSize * 38;
    spacingY =  stanzaHeight * 2;
    drawLattice(parseInt(mouseX / 3, 10), parseInt(mouseY / 3, 10));
}

function updateWheel(e)

// Dokonanie właściwych zmian jeśli poruszy się koło myszy

{
    var evt, wheel;
    evt = window.event || e;
    wheel = evt.detail ? evt.detail * (-120) : evt.wheelDelta;
    if (wheel > 0)
    {
        changeFontSize(1);
    }
    else
    {
        changeFontSize(-1);
    }
}

function markStanza()

// Umieszczenie koordynatów centralnej strofy w boxie

{
    var textInput = document.getElementById("coords");
    textInput.value = canonical(baseI + parseInt(mouseX / 3, 10)) + ",";
    textInput.value += canonical(baseJ + parseInt(mouseY / 3, 10));
```

```
}

function keyDown(e)

// Obsługa klawiszy

{
    var key = String.fromCharCode(e.keyCode);
    if (key === "a" || key === "A")
    {
        changeFontSize(1);
    }
    else if (key === "z" || key === "Z")
    {
        changeFontSize(-1);
    }
    else if (key === ' ')
    {
        markStanza();
    }

    else if (e.keyCode === 37)
    {
        baseI = canonical(baseI - 1);
        drawLattice(parseInt(mouseX / 3, 10), parseInt(mouseY / 3, 10));
    }
    else if (e.keyCode === 38)
    {
        baseJ = canonical(baseJ - 1);
        drawLattice(parseInt(mouseX / 3, 10), parseInt(mouseY / 3, 10));
    }
    else if (e.keyCode === 39)
    {
        baseI = canonical(baseI + 1);
        drawLattice(parseInt(mouseX / 3, 10), parseInt(mouseY / 3, 10));
    }
    else if (e.keyCode === 40)
    {
        baseJ = canonical(baseJ + 1);
        drawLattice(parseInt(mouseX / 3, 10), parseInt(mouseY / 3, 10));
    }
}

function mouseMove(e)

// Obsługa ruchu myszą

{
```

```
    mouseX = e.clientX;
    mouseY = e.clientY;
    drawLattice(parseInt(mouseX / 3, 10), parseInt(mouseY / 3, 10));
    return false;
}

function mouseClick(e)

//Przeskok do nowego rejonu w razie kliknięcia myszy blisko krawędzi,
przerysowanie

{
    if (mouseX > canvas.width * 2 / 3)
    {
        baseI += parseInt(canvas.width / 3, 10);
    }
    else if (mouseX < canvas.width / 3)
    {
        baseI -= parseInt(canvas.width / 3, 10);
    }
    if (mouseY > canvas.height * 2 / 3)
    {
        baseJ += parseInt(canvas.height / 3, 10);
    }
    else if (mouseY < canvas.height / 3)
    {
        baseJ -= parseInt(canvas.height / 3, 10);
    }
    drawLattice(parseInt(mouseX / 3, 10), parseInt(mouseY / 3, 10));
    return false;
}
function resizeCanvas(e)

// Kiedy zmienia się wielkość okna, zmieniamy wielkość canvasa

{
    var div = document.getElementsByTagName('div')[0];
    canvas.width = div.scrollWidth;
    canvas.height = div.scrollHeight;
    context.strokeStyle = "rgba(0,0,128,0.75)";
    drawLattice(0, 0);
}

function setBase(coords)

// Ustawia podstawowe koordynaty pól jeśli nowa tablica znaków jest poprawna
// W przeciwnym wypadku istniejące wartości baseI i baseJ pozostają bez zmian
```

```
{
        newI = parseInt(coords[0], 10);
        newJ = parseInt(coords[1], 10);
        if (!isNaN(newI) && !isNaN(newJ))
        {
            baseI = newI;
            baseJ = newJ;
        }
}

function setup()

// Działa kiedy strona jest wczytywana.
// Uruchamia canvas, nasłuchiwanie zdarzeń etc

{
    var div, newI, newJ, mouseWheelEvent, params = readCoords();

    if (params.length === 2)
    {
        setBase(params);
    }

    canvas = document.getElementsByTagName('canvas')[0];
    if (!canvas.getContext)
    {
        return;
    }

// Dodaje nasłuchiwanie zdarzeń dla ruchu myszy, kliknięcia, strzałki w dół:

    canvas.onmousemove = mouseMove;
    canvas.onclick = mouseClick;
    document.onkeydown = keyDown;
    mouseWheelEvent = (/Firefox/i.test(navigator.userAgent)) ? "DOMMouseScroll" :
"mousewheel";

    if (document.attachEvent)                          // dla IE i Opery
    {
        document.attachEvent("on" + mouseWheelEvent, updateWheel);
    }
    else if (document.addEventListener)                //dla innych przeglądarek
{
        document.addEventListener(mouseWheelEvent, updateWheel, false);
    }

// Dodaje funkcje tekstowe do kontekstu:
```

```
    context = canvas.getContext('2d');
    CanvasTextFunctions.enable(context);

    changeFontSize(0);
    window.onresize = resizeCanvas;
    resizeCanvas(null);
    markStanza();
}

function go()

// Wywoływana po naciśnięciu entera w boxie nawgacji; aktualizuje koordynaty

{
    var textInput, coordPair, URL;
    textInput = document.getElementById("coords");
    coordPair = textInput.value;
    coordPair = coordPair.split(' ').join('');
    coordPair = coordPair.split(':').join(',');
    setBase(coordPair.split(','));
    drawLattice(0, 0);
}
```

Ilustracja 65: Widok generatora poezji w oknie przeglądarki internetowej.

Załącznik 4:
Hello World (kod źródłowy Shakespeare)

Shakespeare jest językiem ezoterycznym, którego forma naśladuje konstrukcję klasycznego dramatu. Ma on następującą składnię[74]:

Lista postaci to deklaracje zmiennych i stosów w formie imienia i opisu.

Wejście, wyjście, wspólne wyjście w didaskaliach odpowiedzialne są za wywoływanie zmiennych.

Za pomocą odpowiednich rzeczowników i przymiotników można przywoływać wartości w zależności od tego czy dany obiekt jest estetyczny czy nie (!). Stąd *świnia* przyjmuje wartość -1 a *kwiat* 1.

Przymiotnikiem podwajamy wartość rzeczownika (czyli możemy uzyskiwać wielokrotności dwójki). Możemy również odejmować od siebie wartości używając wyrażenia `the difference beetween`. Słowo kluczowe `square` podnosi wartość wyrażenia do kwadratu.

`Speak your mind` zwraca wartość zmiennej w formie znaków.

Stąd zdanie:

```
The difference between the square of a little flower and a smelly dirty pig
```

= (2 x 1)² - (2 x 2 x (-1))

= 8;

[74] za: http://en.wikipedia.org/wiki/Shakespeare_(programming_language) (dostęp 4.08.2014)

Oto kod źródłowy programu zwracającego "Hello World":

```
Romeo, a young man with a remarkable patience.
Juliet, a likewise young woman of remarkable grace.
Ophelia, a remarkable woman much in dispute with Hamlet.
Hamlet, the flatterer of Andersen Insulting A/S.

                    Act I: Hamlet's insults and flattery.
                    Scene I: The insulting of Romeo.

[Enter Hamlet and Romeo]


Hamlet:

You lying stupid fatherless big smelly half-witted coward! You are
as stupid as the difference between a handsome rich brave hero and
thyself! Speak your mind!

You are as brave as the sum of your fat little stuffed misused
dusty old rotten codpiece and a beautiful fair warm peaceful sunny
summer's day. You are as healthy as the difference between the sum
of the sweetest reddest rose and my father and yourself! Speak
your mind!

You are as cowardly as the sum of yourself and the difference
between a big mighty proud kingdom and a horse. Speak your mind.
Speak your mind!

[Exit Romeo]
```

Scene II: The praising of Juliet.

[Enter Juliet]

Hamlet:
Thou art as sweet as the sum of the sum of Romeo and his horse and his black cat! Speak thy mind!

[Exit Juliet]

Scene III: The praising of Ophelia.

[Enter Ophelia]

Hamlet:
Thou art as lovely as the product of a large rural town and my amazing bottomless embroidered purse. Speak thy mind!
Thou art as loving as the product of the bluest clearest sweetest sky and the sum of a squirrel and a white horse. Thou art as beautiful as the difference between Juliet and thyself. Speak thy mind!

[Exeunt Ophelia and Hamlet]

Act II: Behind Hamlet's back.

Scene I: Romeo and Juliet's conversation.

[Enter Romeo and Juliet]

Romeo:
Speak your mind. You are as worried as the sum of yourself and the difference between my small smooth hamster and my nose. Speak your mind!

Juliet:
Speak YOUR mind! You are as bad as Hamlet! You are as small as the difference between the square of the difference between my little pony and your big hairy hound and the cube of your sorry little codpiece. Speak your mind!

[Exit Romeo]

Scene II: Juliet and Ophelia's conversation.

[Enter Ophelia]

Juliet:
Thou art as good as the quotient between Romeo and the sum of a small furry animal and a leech. Speak your mind!

Ophelia:
Thou art as disgusting as the quotient between Romeo and twice the difference between a mistletoe and an oozing infected blister! Speak your mind!

[Exeunt]

Bibliografia:

Aarseth E., (1997), *Cybertext. Perspectives on ergodic literature*, The Johns Hopkins University Press, Baltimore.

Arystoteles, (1990), *Dzieła wszystkie (Tom I)*, Państwowe Wydawnictwo Naukowe, Warszawa.

Berry D.M., (2011a), *The computational: Thinking about the digital humanities,* Culture Machine vol. 12.

Berry D.M., (2011b), *The philosophy of software. Code and mediation in digital age*, Palgrave Macmillan, London.

Biecek P., (2014), *Przewodnik po pakiecie R,* Oficyna Wydawnicza GiS, Wrocław.

Bolter J.D., (1990), *Człowiek turinga,* Państwowy Instytut Wydawniczy, Warszawa.

Brookshear J.G., (2012), *Computer science. An overview,* Addison-Wesley, Boston.

Bunnin N., Yu J., (2004), *The Blackwell dictionary of western philosophy,* Blackwell Publishing, Oxford.

Castells M., (2003), G*alaktyka internetu. Refleksje nad internetem, biznesem i społeczeństwem,* Rebis, Poznań.

Celiński P., (2010), *Interfejsy. Cyfrowe technologie w komunikowaniu,* Wydawnictwo Uniwersytetu Wrocławskiego, Wrocław.

Celiński P., (2011), *Medialab: Silnik nowego alfabetyzmu,* [w:] Tarkowski A., Filiciak M., Jałosińska A. (red.), *Medialab. Instrukcja obsługi,* Fundacja Ortus, Chrzelice.

Chaitin G., (2005a)*, Epistemology as informatuon theory: From leibniz to ω,* paper presented at the E-CAP'05 European Computing and Philosophy Conference, Västerås.

Chaitin G., (2005b), *Omega and why maths has no toes,* +Plus Magazine, http://plus.maths.org/content/os/issue37/features/omega/index.

Cheney-Lippold J., (2011), *A new algorithmic identity: Soft biopolitics and the modulation of control,* Theory, Culture and Society 28(6).

Chomsky N., (1992), *Zagadnienia teorii składni,* Zakład Narodowy im. Ossolińskich, Wrocław/Warszawa.

Chun W., (2011), *Programmed visions. Software and memory,* MIT Press, Cambridge/ London.

Cover T., Thomas J., (1991), *Elements of information theory,* John Wiley & Sons, New York.

Dodge M., Kitchin R., (2005), *Code and transduction of space,* Annals of the Association of American Geographers 95(1).

Drucker J., (2009), *Speclab. Digital aesthetics and projects in speculative computing,* University of Chicago Press, Chicago/London.

Eco U., (2013), *Poszukiwanie języka doskonałego,* Aletheia, Warszawa.

Filiciak M., (2008), *Zajrzeć pod powierzchnię ekranu,* Kultura Popularna 4(22).

Floridi L., (1999), *Philosophy and computing,* Routledge, London/New York.

Floridi L., (2013), *Semantic conceptions of information,* Stanford Encyclopedia of Philosophy, http://plato.stanford.edu/entries/information-semantic/#1.2

Fossa L., Prado C., (2007), *Digital culture: The jump form the nineteenth to twenty-first century.* [w:] Rimini F. d. (red.), *A handbook for coding cultures,* d/Lux/MediaArts and Campbelltown Arts Centre, Sydney.

Fortescue M., (2001), *Pattern and process: A whiteheadian perspective on linguistics,* John Benjamins Publishing Company, Amsterdam/Philadelphia.

Frege F. G., (1979), *Begriffsschrift, eine der arithmetischen nachgebildete formelsprache des reinen denkens,* Verlag von Louis Nebert, Halle am See.

Fuller M., (2003), *Behind the blip. Essays on the culture of software,* Autonomedia, New York.

Fuller M. (red.), (2008), *Software studies. A lexicon*, MIT Press, London/Cambridge.

Fuller M., Matos S., (2011), *Fcj-135 feral computing: From ubiquitous calculation to wild interactions,* The Fibreculture Journal(19).

Galloway A., (2004), *Protocol. How control exists after decentralization,* MIT Press, Cambridge/London.

Galloway A., (2008), *Język chce, aby go nie dostrzegać,* Kultura Popularna 4(22).

Gierasimczuk N., (2010), *Psychologiczna adekwatność modelu identyfikowalności granicy.* [w:] Urbański M., Przybysz P. (red.), *Funkcje umysłu,* Zysk i Ska, Poznań.

Gillespie T., Boczkowski P., Foot K. (red.), *Media technologies. Essays on communication, materiality and society,* (2014), MIT Press, Cambridge/London.

Goffey A., (2008), *Algorithm,* [w:] Fuller M. (red.), *Software studies. A lexicon,* MIT Press, London/Cambridge.

Graham P., (2004), *Hakerzy i malarze. Wielkie idee ery komputerów,* Helion, Gliwice.

Greenfield A., (2006), *Everyware. The dawning age of ubiquitous computing,* New Riders, Berkeley.

Halavais A., (2009), *Search engine society,* Polity Press, Cambridge/Malden.

Harel D., (2000), *Rzecz o istocie informatyki. Algorytmika,* Wydawnictwa Naukowo-Techniczne, Warszawa.

Ifrah G., (2006), *Historia powszechna cyfr (Tom 1),* Wydawnictwo WAB, Warszawa.

Kaisler S., (2005), *Software paradigms,* John Wiley & Sons, Hoboken.

Kant I., (1957), *Krytyka czystego rozumu,* Państwowe Wydawnictwo Naukowe, Warszawa.

Kavanaugh L., (2007), *The philosophical foundations of network theory,* paper presented at the New Network Theory, Amsterdam.

Keringhan B., Pike R., (2011), *Lekcja programowania. Najlepsze praktyki,* Helion, Gliwice.

Kittler F., (1997), *Essays. Literature, media, information systems,* Routledge, New York.

Kowalski R., (1979), *Algorithm = logic + control,* Communications of the ACM.

Kranenburg R. v., (2008), *The internet of things. A criticque of ambient technology and the all-seeing network of RFID,* Institute for Network Cultures, Amsterdam.

Kuhn T., (1968), *Struktura rewolucji naukowych,* Państwowe Wydawnictwo Naukowe, Warszawa.

Kuriata E., (2001), *Teoria informacji i kodowania,* Oficyna Wydawnicza Politechniki Zielonogórskiej, Zielona Góra.

Law J., (1999), *After ant: Topology, naming and complexity,* [w:] Hassard J., Law J. (red.), *Actor network theory and after,* Blackwell, Oxford.

Leibniz G.W, (1951), *The art of discovery,* [w:] Wiener P. (red.), *Leibniz: Selections.* Charles Scribner's Sons, New York.

Leibniz G.W., (1955), *Nowe rozważania dotyczące rozumu ludzkiego*, Państwowe Wydawnictwo Naukowe, Warszawa.

Levy P., (2005), *Drugi potop.* [w:] Hopfinger M. (red.), *Nowe media w komunikacji społecznej XX wieku,* Oficyna naukowa, Warszawa.

Locke J., (1955), *Rozważania dotyczące rozumu ludzkiego,* Państwowe Wydawnictwa Naukowe, Warszawa.

Mackenzie A., (2002), *Transductions. Bodies and machines at speed,* Continuum, London/ New York.

Mackenzie A., (2003a), *The problem of computer code: Leviathan or common power,* http://www.lancs.ac.uk/staff/mackenza/papers/code-leviathan.pdf.

Mackenzie A., (2003b)*, Transduction: Invention, innovation and collective life,* http://www.lancaster.ac.uk/staff/mackenza/papers/transduction.pdf.

Manovich L., (2006), *Język nowych mediów,* Wydawnictwa Akademickie i Profesjonalne, Warszawa.

Manovich L., (2013), *Software takes command,* Bloomsbury Academic, New York/London.

Marino M., (2006), *Critical code studies. Electronic Book Review,* http://www.electronicbookreview.com/thread/electropoetics/codology

Myoo S., (2013), *Ontoelektronika,* Wydawnictwo Uniwersytetu Jagiellońskiego, Kraków.

Nacher A., (2013), *Poza cyfrowość w zwrocie cyfrowym - od humanistyki cyfrowej do spekulatywnej komputacji,* [w:] Bomba R., Radomski A. (red.), *Zwrot cyfrowy w humanistyce,* E-naukowiec, Lublin.

Nadin M., (2007), *Semiotic machine,* The Public Journal of Semiotics I(1).

Nalepa G. J., (2011), *Pojęcia w inżynierii wiedzy,* [w:] Chuderski A., Bremer J. (red.), *Pojęcia. Jak reprezentujemy i kategoryzujemy świat,* Universitas, Kraków.

Montfort N., Badouin P., Bell J., Bogost I., Douglass J., Marino M., Mateas M., Reas C., Sample M., Vawter N., (2012), *10 print chr$(205.5+rnd(1)); : Goto 10,* http://trope-tank.mit.edu/10_PRINT_121114.pdf

Peckhaus V., (2005). *Calculus ratiocinator vs characteristica universalis? The two traditions in logic, revisited.* [w:] Reck E., Beaney M. (red.), *Gottlob Frege. Critical assessments of leading philosophers*, Routledge, Oxford.

Petzold C., (2002), *Kod. Ukryty język sprzętu komputerowego i oprogramowania,* Wydawnictwa Naukowo-Techniczne, Warszawa.

Porębski B., Palacz W., Grabska E., Grabska-Gradzińska I., (2012), *Towards a graph-based model of computer games,* paper presented at the Proc. of Int. Conf. on Advances in Comminication and Information Technology.

Poslad S., (2009), *Ubiquitous computing. Smart devices, environments and interactions,* John Wiley and Sons, London.

Preimer R., (1991), *Introductory signal processing,* World Scientific, London.

Quine W. v. O., (2006), *Korzenie ontologii,* Aletheia, Warszawa.

Rabinow P., (1999), *French DNA: Trouble in purgatory,* University of Chicago Press, Chicago.

Reale G., (2005), *Historia filozofii starożytnej (Tom 5),* Wydawnictwo KUL, Lublin.

Roy P. v., Haridi S., (2004), *Concepts, techniques and models of computer programming,* MIT Press, London/Cambridge.

Sack W., (2008), *Memory,* [w:] Fuller M. (red.), *Software studies. A lexicon,* MIT Press, London/Cambridge.

Schmuller J., (2003), *UML dla każdego*, Helion, Gliwice.

Shannon C., Weaver W., (1980), *The mathematical theory of communication,* University of Illinois Press, Chicago.

Szkudlarek B., (2007), *Actor-network theory - ontologizing realities,* paper presented at the New Network Theory, Amsterdam.

Tufte E., (2006), *Beautiful evidence,* Graphic Press, Cheshire CT.

Weisfeld M., (2010), *Myślenie obiektowe w programowaniu,* Helion, Gliwice.

Wittgenstein L., (2000), *Tractatus logico-philosophicus,* Wydawnictwo Naukowe PWN, Warszawa.

Writh N., (2000), *Algorytmy + struktury danych = programy,* Wydawnictwa Naukowo-Techniczne, Warszawa.

Wróblewski P., (2010), *Algorytmy, struktury danych i techniki programowania.* Helion, Gliwice.