Dalle competenze alla consapevolezza digitale:
capire la complessità e la non neutralità del software

Silvia Crafa - Università di Padova

Computer science is no more about computers
than astronomy is about telescopes.
–Edsger Dijkstra

1 Introduzione

L'informatica viene spesso identificata con la produzione e l'uso di tecnologie digitali, che vanno da artefatti immateriali come gli algoritmi e il software, fino alle architetture hardware e le reti fisiche di comunicazione. Certamente gli aspetti tecnologici fanno parte dell'informatica, e il suo lato ingegneristico si occupa proprio della progettazione, dello sviluppo e della gestione di sistemi tecnologici, dall'analisi dei costi e benefici alla messa in atto di questi sistemi. Ma l'informatica ha anche un aspetto su cui ci si sofferma di meno: quello logico-matematico, prettamente scientifico; è ciò che in inglese si identifica con il termine *computer science* in contrapposizione a *information and communication technology*. Il linguaggio matematico e il rigore del formalismo logico sono gli strumenti con cui si esprime la scienza informatica. Questi strumenti sono necessari per definire e analizzare con precisione il comportamento dei sistemi tecnologici, ma allo stesso tempo ne definiscono rigidamente la struttura e il senso. Quando dunque, come oggi, le tecnologie digitali sono pervasive nella nostra vita quotidiana, diventa importante conoscere il linguaggio della scienza informatica, proprio per poter essere consapevoli di come esso delimita e definisce le tecnologie che produce.

Pensiamo ad esempio allo studio di fenomeni sociali complessi, come l'evoluzione di una pandemia. L'approccio fisico- matematico prevede l'uso di modelli e simulazioni che permettono di comprendere meglio il fenomeno, fare ipotesi e verificare previsioni. Ma la scienza informatica aggiunge la

capacità di computare a grana fine il comportamento di un modello, aggiungendo concretezza e dettaglio alla simulazione, anche grazie all'integrazione di grandi quantità di dati[1]. L'approccio informatico rappresenta dunque uno strumento molto potente per migliorare la comprensione di un fenomeno complesso, ma allo stesso tempo è fondamentale tenere presente che questi modelli e la loro computazione si basano su assunzioni e metodologie che, pur essendo necessarie per la formalizzazione logico-matematica, hanno degli effetti non neutrali sui risultati ottenibili e sul loro senso. Passare da un fenomeno reale ad un insieme di dati che lo rappresenta comporta uno scarto che può essere significativo; il tipo di operazioni logico-matematiche che verranno fatte sui dati (algoritmo), forzano a rappresentare la realtà in un formato preciso. Ad esempio, consideriamo le operazioni più semplici, che sono quelle numeriche, come la previsione di un valore massimo o il calcolo di un valore medio; se un algoritmo usa queste operazioni, significa che i dati del problema devono essere dei numeri, e dunque che il fenomeno analizzato deve essere quantificato e misurato. La gestione della pandemia di COVID-19 ha messo in luce quanto sia difficile fornire un valore corretto anche per dati direttamente quantificabili, come il numero di soggetti malati o il numero di posti disponibili in terapia intensiva; ancora più problematico è tradurre in una quantità numerica fenomeni come l'indice di trasmissibilità del virus, o il livello di disagio psicologico della popolazione. Il problema dei dati diventa perfino cruciale negli algoritmi di intelligenza artificiale, in cui criticità come discriminazione e opacità dei modelli di *deep learning* sono temi su cui sta emergendo sempre più attenzione pubblica[2].

Senza bisogno di addentrarsi nell'intelligenza artificiale, anche la semplice produzione di software di tipo tradizionale presenta caratteristiche di cui è necessario diventare consapevoli. Solo in questo modo sarà possibile decidere al meglio quale ruolo affidare al software, scegliendo accuratamente quando usarlo, come usarlo, e come integrarlo in uno specifico contesto aziendale, sociale o umano. In questo articolo ci occuperemo proprio del software, osservando che, se la scrittura del software è competenza propria degli informatici, un'adeguata cultura digitale richiede ad ogni professionista e cittadino di essere consapevole del grado di complessità e non neutralità insite nella produzione dei programmi. È importante comprendere i diversi livelli

[1] Henzingher, Denning 2009.

[2] Pelillo, Scantamburlo 2021; Scantamburlo, Charlesworth, Cristianini 2019; Crawford 2021.

di astrazione coinvolti nello sviluppo e nell'esecuzione del software, capire cosa significa e che limiti ha la valutazione della qualità di un software; significa infine riflettere su cosa significhi e quanto sia complesso comunicare (le caratteristiche di) un software.

2 Il pensiero computazionale

Da qualche anno si usa il termine pensiero computazionale[3] per indicare ciò che caratterizza l'informatica rispetto ad altre scienze, e lo si usa come guida nella costruzione di percorsi formativi, soprattutto scolastici, in questa disciplina. Pur non essendoci una definizione univoca di questo termine, il pensiero computazionale caratterizza il modo con cui pensa un informatico quando affronta un problema. Nella definizione di Enrico Nardelli[4], si tratta della

> capacità di formulare problemi in modo che ammettano delle soluzioni rappresentabili in formato eseguibile da un uomo o da una macchina, o da una combinazione dei due (information processing agent).

Questa definizione contiene diversi elementi importanti. Più sottilmente del cosiddetto *problem solving*, il pensiero computazionale orienta la formulazione di un problema verso la sua soluzione. Questo approccio già imposta un problema, tramite la sua formulazione, non tanto verso una sua analisi o comprensione generale, ma verso una sua soluzione operativa. Non tutti i problemi scientifici ammettono una **soluzione operativa**, e la teoria della computabilità, nata con Alan Turing, è proprio quella branca dell'informatica che si occupa di studiare con precisione quali sono i limiti delle classi di problemi che ammettono una soluzione operativa (i.e., computabile), lasciando alla matematica e alla logica la trattazione di quei problemi la cui soluzione non è computabile[5]. Tralasciando gli aspetti teorici, ciò che conta qui osservare è l'enfasi dell'informatica sulla produzione di soluzioni effettive. Altro elemento importante è proprio ciò che caratterizza una soluzione effettiva: il fatto che sia scritta in un formato eseguibile da un agente automatico o automatizzabile. Un programma informatico, un software, consiste dunque in

[3] Wing 2006.
[4] Nardelli 2019a; Nardelli 2019b.
[5] Cutland 1980.

una serie di istruzioni che spiegano come effettuare una computazione, e.g., un calcolo matematico, un'elaborazione di un testo, un'operazione grafica, la riproduzione di un video. Queste istruzioni non sono altro che frasi di un qualche **linguaggio di programmazione**, progettato per essere un linguaggio conciso e non ambiguo; che ammette cioè poche istruzioni, ciascuna con un significato non ambiguo e indipendente dal contesto.

L'uso di linguaggi di programmazione è il fondamento su cui si basa l'esecuzione automatica effettuata dai computer. D'altra parte, la necessità di rappresentare la soluzione di un problema in questo formato eseguibile automaticamente, esclude molta dell'espressività del linguaggio naturale, considerando preciso solo ciò che è formalizzabile in uno specifico linguaggio di programmazione. Inoltre, l'uso di linguaggi formali non evita l'introduzione di errori: i più semplici sono gli **errori sintattici**, che non rispettano l'ortografia del linguaggio, e sono generalmente individuati dalla macchina nel momento in cui si trova ad eseguire un'istruzione che non riconosce. Molto più sottili sono gli **errori di semantica** (o errori logici), che rappresentano i casi in cui il programmatore ha inserito nel software delle istruzioni sintatticamente corrette, ma che istruiscono la macchina a produrre un risultato errato rispetto alla soluzione che intendeva rappresentare. Ad esempio, data una mappa stradale, consideriamo un programma che risolve il problema di calcolare la strada più breve per raggiungere una certa destinazione. Un software che propone un percorso che passa due volte per una stessa città intermedia è un programma che presenta un errore semantico. Questo tipo di errori può dipendere da un errore di ragionamento del programmatore, che in effetti non ha individuato la soluzione corretta del problema, oppure, ancora più sottilmente, il ragionamento logico era corretto ma l'errore semantico è stato introdotto nel momento in cui la soluzione (corretta) viene rappresentata tramite le istruzioni del linguaggio di programmazione. Individuare gli errori logico-semantici in un software può essere estremamente difficile, soprattutto quando il problema sta nel disallineamento tra ciò che si vuole esprimere, che chiameremo la **specifica di un programma**, e ciò che effettivamente si è scritto tramite una sequenza di istruzioni, che chiameremo **implementazione del programma**. L'ampiezza di questo disallineamento dipende essenzialmente dalla diversità tra il potere espressivo del linguaggio di programmazione e quello del linguaggio naturale (o altri linguaggi di specifica). Inoltre gli errori semantici possono rivelarsi solo nel caso di particolari input, ad esempio il programma citato sopra può suggerire un percorso errato solo per alcune destinazioni.

Consideriamo come esempio il caso degli *smart contracts*, cioè programmi che vengono eseguiti automaticamente su blockchain, cioè su una sorta di computer globale decentralizzato il cui funzionamento è garantito dal controllo contemporaneo di migliaia di utenti. Gli smart contracts sono la soluzione più efficace quando due persone che non si fidano l'una dell'altra si accordano su una certa azione, ad esempio una scommessa associata ad una partita di calcio e la modalità di riscossione della vincita, e affidano l'esecuzione automatica di questa azione al codice di un programma. Possiamo dire in questo caso che la scommessa rappresenta la specifica del contratto sottoscritto, ma ciò che viene effettivamente eseguito è la corrispondente implementazione, cioè il codice vero e proprio dello smart contract. Per dirimere le questioni che possono sorgere in caso di disallineamento tra la specifica e l'implementazione, ad esempio un comportamento non voluto causato da un errore logico nella programmazione, il mondo delle blockchain si basa sul principio *code-is-law*: è l'implementazione del programma (pubblicamente disponibile) che fa fede nell'accordo tra le parti, non la specifica. Nella pratica però questo non risolve le controversie, poiché quando sono in gioco volumi di denaro elevati nessuno è davvero disponibile a considerare un *bug* di programmazione come parte del contratto che aveva sottoscritto[6]. L'indotto economico creato dall'uso degli smart contract come strumenti di intermediazione automatica è talmente rilevante che si stanno moltiplicando gli sforzi per inventare dei nuovi linguaggi di programmazione e specifica, che minimizzino il rischio di questi disallineamenti. È ad esempio il caso di Lexon[7]: un linguaggio molto vicino al linguaggio naturale usato per redigere contratti di natura legale, ma che si presta ad essere automaticamente tradotto nel codice sorgente di uno smart contract. Questo offre una certa garanzia che l'implementazione sia corretta rispetto alla specifica data, ma non elimina il problema di fondo, perché la semantica dei linguaggi che operano su blockchain è particolarmente complessa.

Riprenderemo questo argomento in una sezione successiva, sottolineando ora solamente come la programmazione sia lo strumento concreto per allenare ed esercitare la specifica capacità di *problem solving* che distingue il pensiero computazionale. La programmazione fa dunque parte di una serie di strumenti mentali che caratterizzano questo tipo di pensiero, e che con-

[6]Spode 2017.

[7]http://www.lexon.tech/

sentono di passare da un'idea alla sua realizzazione effettiva. In questo senso il pensiero computazionale è caratterizzato da un'attitudine all'esplorazione creativa delle possibili soluzioni di un problema, e da un approccio di tipo *trial-and-error* che porta il programmatore a sperimentare ed ideare una soluzione contemporaneamente alla sua formulazione in un software. Quel che si fa è abbozzare un *prototipo* di programma, che poi si raffina e si corregge con passaggi successivi. Questo tipo di metodologia, creativa e sperimentale, è davvero molto frequente; ha il vantaggio di offrire rapidamente dei primi risultati operativi, ma può rendere più difficile la valutazione della qualità finale del software, che tende a mescolare in un tutt'uno la specifica—dunque la logica di funzionamento- e l'implementazione del programma.

3 La differenza tra algoritmo e software

I termini algoritmo e software vengono spesso usati come sinonimi, ma è importante avere chiara la loro differenza, perché si riferiscono a concetti diversi, e proprio nella loro differenza si nascondono degli aspetti rilevanti, che sono spesso poco in luce, ma di cui è bene essere consapevoli. Un algoritmo rappresenta la logica di funzionamento di un programma, mentre il software è il codice di un programma scritto in un preciso linguaggio di programmazione ed eseguibile da un computer. Un algoritmo è definito come "una sequenza non ambigua di passi che determina la procedura di soluzione di un problema". Ma questa sequenza di passi può essere scritta in qualsiasi linguaggio: in genere si usa uno pseudo-codice, che sta a metà strada tra linguaggio naturale, matematica e linguaggio di programmazione, in modo che sia sufficientemente preciso e non ambiguo, senza però conformarsi a rigide regole sintattiche. Un esempio classico è l'algoritmo di Euclide, che definisce un procedimento matematico per trovare il massimo comune divisore tra due numeri interi. Se ad uno studente basta leggere la descrizione di questa procedura per saper calcolare il massimo comune divisore, per un computer servono delle istruzioni più specifiche. L'algoritmo va cioè implementato, tradotto in un software scritto in un linguaggio di programmazione sintatticamente preciso: solo questa traduzione è pienamente non ambigua ed effettivamente eseguibile.

Riassumendo, abbiamo già introdotto tre distinti livelli di astrazione: la **specifica** di un programma, e.g., "il programma P prende in input due numeri interi e restituisce in output il loro massimo comune divisore", l'**algoritmo**

(o logica di funzionamento), e.g., "il programma P calcola il massimo comune divisore seguendo l'algoritmo di Euclide", e l'**implementazione**, e.g., il codice in linguaggio Python che implementa l'algoritmo di Euclide. Il livello dell'algoritmo è dedicato ad esprimere un'idea risolutiva, ma essendo piuttosto astratto, può essere molto meno preciso di quanto si possa immaginare, un po' come accade come per il progetto di una casa fatto da un architetto, che deve poi essere "implementato" in una casa vera e propria. Lo stesso algoritmo si può a volte implementare in modi molto diversi, scegliendo modi diversi di rappresentare i dati, scegliendo linguaggi e tecnologie di programmazione più o meno opportuni, e scegliendo come trattare ogni specifico dettaglio. Ad esempio, un algoritmo può parlare di operazioni da effettuare su un insieme di numeri, ma il concetto matematico di *insieme* si può rappresentare in un programma software con diversi tipi di strutture dati (e.g., array, stack, liste, alberi, stream infiniti). L'area dell'informatica che si occupa degli algoritmi insegna proprio a fare queste scelte di implementazione, e a gestire la differenza, a volte notevole, tra l'efficienza (complessità) di un algoritmo teorico e quella delle sue implementazioni.

La differenza tra algoritmo e software è particolarmente rilevante anche quando si vuole comprendere **quando un programma è sbagliato**, al fine di valutare ad esempio gli effetti legali del suo impiego. Consideriamo ad esempio un software che determina la graduatoria di ammissione ad un corso di laurea a numero programmato. L'algoritmo sottostante potrebbe utilizzare delle regole di posizionamento discriminatorie, fedelmente tradotte nel software. In questo caso il software è corretto, mentre il problema sta nell'algoritmo. Consideriamo inoltre il caso in cui l'algoritmo non dia alcuna indicazione su come trattare i casi di pari merito (i.e., la specifica non è completa). Il programma che implementa l'algoritmo è per definizione non ambiguo: le istruzioni dei linguaggi di programmazione sono fatte in modo che la macchina sappia come comportarsi in ogni situazione, incluse le situazioni a cui il programmatore non ha esplicitamente pensato. Consideriamo ad esempio le seguenti tre istruzioni condizionali, che sono molto simili tra loro ma che possono produrre conseguenze molto diverse[8]:

- *se x è maggiore di y allora lo studente è ammesso:*
  `if (x > y) then ammesso=true`,

[8] la prima istruzione è equivalente alla seconda oppure alla terza a seconda di come è inizializzata la variabile ammesso, cioè a seconda di come è scritto il resto del programma.

- *se x è maggiore di y, lo studente è ammesso, altrimenti no:*
  `if (x > y) then ammesso=true else ammesso=false`,
- *se x è maggiore o uguale a y allora lo studente è ammesso, altrimenti no:*
  `if (x>=y) then ammesso=true else ammesso=false`.

Piccole differenze nelle istruzioni, come queste, possono finire per gestire implicitamente i casi di pari merito della graduatoria di ammissione, anche senza l'esplicita intenzione del programmatore, che si è limitato ad implementare l'algoritmo senza accorgersi che la specifica non era completa.

Questo esempio fa capire quanto sia complesso descrivere in maniera precisa ed esaustiva cosa fa (davvero) un programma e come lo fa. Avere a disposizione il codice sorgente offre completa trasparenza, ma non completa intelligibilità. Piccoli *bug*, ma anche stili poco chiari di programmazione, rendono difficile la lettura del codice. Inoltre ripercorrere passo passo la sequenza di istruzioni che conducono da un input al corrispondente output (analisi del **flusso di controllo**) è un approccio impraticabile già in programmi di media complessità, soprattutto quando si introducono elementi di parallelismo e comunicazioni remote, come succede comunemente con il software che viene eseguito su *browser*, su *smartphone* (*app*) o su piattaforma *cloud*. E se già l'analisi della correttezza del software rispetto alla sua specifica presenta queste difficoltà, tantopiù sarà difficile la valutazione della qualità del software rispetto a requisiti più complessi e difficilmente traducibili in codice, come la non discriminazione, l'eticità, il rispetto della privacy e dell'autonomia dell'utente, o l'appropriatezza al suo contesto d'uso.

4 Il processo di sviluppo, la vita e la qualità del software

Un software complesso, e.g., il gestionale di un'azienda, l'infrastruttura digitale di una banca, il software che guida un satellite o un social network, ha un ciclo di vita che prevede una fase iniziale di sviluppo ma anche un continuo ciclo di uso e manutenzione, per effettuare aggiornamenti che ne correggono o ne estendono le funzionalità. La progettazione del software deve quindi tenere conto dell'intero ciclo di vita previsto, in modo da scrivere

fin dall'inizio codice non solo corretto ma anche più facilmente integrabile e manutenibile.

I principi di progettazione, costruzione, stima dei costi, gestione del ciclo di vita, sono gli argomenti di cui si occupa la branca dell'informatica chiamata ingegneria del software. Il cardine che regge questa complessità è la produzione di **documentazione**, chiara e precisa, dell'intero processo di costruzione del software. Esistono numerosissimi strumenti, tecniche e linee guida per la produzione della documentazione, ma ciononostante non esistono notazioni stabili, standard e pienamente efficaci. Profonde conseguenze derivano dalla difficoltà intrinseca di *comunicare* in modo esaustivo (*i*) i requisiti richiesti, (*ii*) le caratteristiche interne e le modalità di manutenzione, (*iii*) le modalità di utilizzo da parte degli utenti, (*iv*) la qualità del software, in termini di assenza di errori, accuratezza rispetto ai requisiti, performance, cyber-sicurezza, usabilità. Il caso della raccolta dei requisiti del software è emblematico, perché richiede agli esperti informatici di mettersi in dialogo con gli *stakeholder*, che non hanno necessariamente competenze informatiche, come diversi tipi di utenti, il responsabile di aspetti legali (e.g., per la privacy dei dati), o economici. Questa fase richiede di identificare i servizi che dovranno essere forniti dal sistema software, trovando dei compromessi tra i desideri dei committenti, i bisogni effettivi, i costi sostenibili e la fattibilità concreta. Le linee guida per la raccolta dei requisiti indicano come compilare un preciso **documento di specifica dei requisiti del software**, che costituisce a tutti gli effetti un accordo tra le parti: guida i programmatori allo sviluppo del programma e si usa per dirimere eventuali controversie. Usando la terminologia introdotta in precedenza, questo documento rappresenta dunque la *specifica* ufficiale del programma, e una volta redatto con accuratezza sembra che il problema sia solo realizzare un'*implementazione* che rispetti esattamente la specifica data. Nella pratica, invece, accade molto spesso che i requisiti raccolti non siano stati pienamente compresi, o siano incompleti, richiedendo quindi modifiche o integrazioni tramite successive revisioni che coinvolgono le parti interessate, come ad esempio una richiesta tardiva di aggiungere delle funzionalità in più al sito web aziendale. Tutto questo accade talmente spesso che l'ingegneria del software prevede principi e le linee guida proprio per la gestione dell'aggiornamento e la correzione dei requisiti (i.e., della specifica del software), in modo da ridurre i costi aggiuntivi, i ritardi e gli errori che facilmente si possono generare nel corrispondente aggiornamento dell'implementazione.

Abbiamo già descritto come sia complesso da un lato specificare esattamente cosa vogliamo che faccia un programma, e dall'altro garantire che il programma si comporta esattamente come richiesto, senza errori di programmazione (sintattici e semantici) e rispettando tutti i requisiti dati. In termini più generali, si tratta del problema della **qualità del software**. Anche qui non mancano le linee guida e le buone prassi per effettuare dei test esaustivi sul codice sorgente e sul comportamento del programma, così come esistono degli standard di qualità per il processo di sviluppo del software, in analogia con quanto succede in altre branche dell'ingegneria. Però a differenza dei macchinari meccanici o delle strutture edili, i software complessi possono avere un numero talmente elevato di diverse esecuzioni possibili che non è praticabile testarle tutte. Per comprendere questa complessità del testare il comportamento del software, va sottolineato che i programmi di medie e grandi dimensioni si descrivono meglio con il termine **sistemi software**: sono composti cioè da diversi programmi che interagiscono tra di loro, scritti in modo indipendente, e che richiedono uno specifico ambiente di esecuzione. Pensiamo ad esempio al software che gestisce il contenuto di un sito web: dovrà necessariamente interagire con il *browser*, che è a sua volta un programma (e.g., Chrome di Google, Internet Explorer di Microsoft, Firefox della Mozilla Foundation) che permette di visualizzare i siti e trasmettere i dati del web utilizzando la rete di comunicazione Internet; ma il software del sito potrà interagire anche con il software di un servizio esterno di pagamento digitale, come PayPal o PagoPA, oppure interagire con il sistema operativo del computer o dello smartphone che si sta utilizzando, al fine di usufruire della fotocamera. È dunque intuibile come sia difficile assicurare l'assenza di errori in uno scenario del genere. Ad esempio, se volessimo testare tutte le esecuzioni possibili del software Chrome, dovremmo testare tutte le sue interazioni con il software di ogni possibile sito web, cosa che è chiaramente impraticabile. Il sito web "Collection of software bugs"[9] è uno dei tanti che raccoglie informazioni sui maggiori disastri che si sono verificati a causa di errori software, come errori di arrotondamento dei numeri usati nei calcoli, oppure l'uso del carattere '.' invece del carattere ',' all'interno di una istruzione del programma, oppure la scorretta interazione tra parti diverse di un sistema software.

La ricerca informatica lavora moltissimo sulla certificazione della qualità del software, sia con approcci applicativi, come gli strumenti di *testing* e

[9]https://www5.in.tum.de/ huckle/bugse.html

debugging, sia con approcci teorico-matematici, che mirano ad offrire delle dimostrazioni formali dell'assenza di specifici errori nei programmi, e.g., errori di arrotondamento nei calcoli o specifici attacchi di sicurezza. Resta comunque un problema intrinseco, cioè cosa sia un software di qualità e come si possa *misurare* la qualità del software. La qualità di un programma è indissolubilmente legata ai suoi requisiti, sulle cui criticità abbiamo già discusso. Inoltre, se pensiamo ai programmi che effettuano delle decisioni come la concessione di un mutuo, la preselezione del personale, o che decidono che tipo di informazione presentare nella bacheca personale di un social network, allora la scelta dei requisiti diventa anche una scelta su quali valori sociali e quale tipo di comunità si vuole promuovere. Come ha sottolineato J.C De Martin all'apertura della prima edizione della Biennale Tecnologia del 2020[10], la tecnologia è profondamente umana, non nasce spontaneamente ma è espressione di valori e di visioni del mondo, che inevitabilmente si riflettono nelle scelte dei requisiti che guidano la progettazione del software.

5 Non neutralità dei sistemi digitali

Al giorno d'oggi i sistemi software sono costituiti da un'interazione complessa tra persone e tecnologia: la progettazione tecnica tiene conto delle necessità e dei comportamenti degli utenti, ma a sua volta retroagisce su questi comportamenti, incentivandone alcuni e limitandone altri. L'esempio più discusso sono i social network, e il loro algoritmo che seleziona i contenuti visibili dagli utenti, generando effetti di polarizzazione delle opinioni noti come *filter bubbles* e *echo chambers*[11]. Ma hanno effetti sui comportamenti delle persone anche applicazioni come il calendario digitale o il navigatore integrato nel computer di bordo dell'auto. La specificità dell'interfaccia di uno strumento di videoconferenza, come ad esempio Zoom, può determinare il tipo di didattica a distanza che viene progettata: ad esempio avere o meno la possibilità di suddividere i partecipanti in stanze distinte, abilita o meno la discussione in sottogruppi.

Gli effetti di un programma sui comportamenti degli utenti possono essere stati progettati esplicitamente, come accade per l'inserimento nell'interfaccia di elementi che generano dipendenza, in modo da massimizzare il tempo speso dall'utente sulla piattaforma e monetizzarlo, secondo i principi dell'*Attention*

[10] http://demartin.polito.it/tecnologiaeumanita

[11] Davies 2020.

Economy[12]. In altri casi gli effetti sugli utenti possono essere il frutto implicito di una progettazione non accurata. Se pensiamo ai sistemi di prenotazioni delle prestazioni sanitarie, come una vaccinazione, chiaramente l'usabilità del software da parte degli utenti è uno dei requisiti cardine, ma se non si tiene presente che esistono diversi tipi di utenti, il software realizzato può rivelarsi non accessibile a specifiche categorie di persone, finendo per esempio per mettere in difficoltà gli anziani o chiunque non possieda un indirizzo email[13].

Dunque prima ancora di addentrarsi in discussioni molto generali come l'etica degli strumenti digitali, è importante sviluppare uno sguardo non ingenuo su come il software viene progettato, individuandone i caratteri di non neutralità. Il modo giusto di guardare ai sistemi digitali di oggi, è quello di intenderli sempre come **sistemi socio-tecnologici**, andando a svelare lo specifico legame tra progettazione tecnica e comportamento dell'utente, per chiedersi poi a chi serve questa applicazione? Che comportamenti incentiva? Che tipo di comunità promuove? Che tipo di valori sta supportando?

Oltre alla progettazione, la non neutralità può stare nelle scelte di implementazione di un sistema software. Un esempio in questo senso è offerto dalla differenza tra la comunicazione tramite email e la comunicazione tramite servizi di messaggistica. Mentre è possibile scambiare email anche tra persone che hanno scelto diversi *provider* del servizio di posta elettronica (e.g., GMail e Yahoo), non è possibile mandare un messaggio da un account WhatsApp ad un account Telegram. Una scelta che si può definire politica, all'inizio degli anni '80, aveva imposto a tutti i provider del servizio email di aderire ad uno stesso standard tecnico di implementazione (cioè di rispettare una certa specifica di requisiti): il protocollo SMTP (*simple mail transfer protocol*), assicurando così l'interoperabilità dei servizi. La stessa scelta non è stata fatta per i servizi di messaggistica o di videoconferenza, incentivando un diverso tipo di economia, in cui i fornitori del servizio tendono a legare a sé gli utenti in modo sempre più stretto in un effetto *lock-in*.

La non neutralità può nascondersi anche nel fondamento scientifico che sta alla base di un software, come accade ad esempio agli algoritmi di intelligenza artificiale. Gli assunti teorici su cui si basa l'inferenza predittiva degli algoritmi di *machine learning*[14], fanno sì che errori o pregiudizi insiti nei dati su cui questi algoritmi vengono allenati, siano amplificati nelle predi-

[12]Lewis 2017; Menczer, Hills 2021; Crafa, Rizzo 2020; Signorelli 2019.
[13]Crafa, Gaggi 2019.
[14]Codenotti, Leoncini 2020; Mitchell 2020.

zioni e nelle raccomandazioni proposte in output. Allo stesso tempo, il loro trattamento di tipo statistico dei dati evidenzia i pattern più frequenti, ma penalizza le divergenze e i casi più originali, che diventano anomalie difficili da comprendere e da gestire per l'algoritmo, che quindi le "normalizza" di fatto scartandole. In senso più ampio, la profonda non neutralità dell'impianto scientifico dell'intelligenza artificiale e le relative conseguenze sul piano sociale sono argomenti su cui si sta ampliando il dibattito sia scientifico che pubblico[15], fino alle proposte di framework normativi come quello recentemente proposto dalla Commissione Europea[16].

Infine stiamo comprendendo che la tecnologia, per quanto immateriale, non è per nulla neutrale in termini di impatto ambientale: i giganteschi *data center* necessari ad esempio al funzionamento di Google Search e Facebook, i servizi *cloud* venduti ad esempio da Amazon Web Services e Microsoft Azure e usati come infrastruttura digitale dalla maggior parte delle aziende e pubbliche amministrazioni del pianeta, ma anche i servizi di *streaming* video di aziende come Netflix e HBO, consumano quantità enormi di energia, per la maggior parte proveniente da fonti non rinnovabili. In aggiunta, tecnologie in rapida crescita il cui sfruttamento è solo all'inizio, come l'intelligenza artificiale, l'*Internet of Things*, la blockchain, le reti 5G, producono sull'ambente un'impronta energetica che diventerà molto presto insostenibile[17].

Chiudiamo questa sezione con un esempio positivo di progettazione non neutrale: Wikipedia, che a gennaio 2021 ha compiuto 20 anni, è rimasta la piattaforma più fedele allo spirito originario di Internet[18], quello cioè di una rete di informazioni a disposizione di tutti, affinché anche le persone comuni potessero usare i loro computer come strumenti di liberazione, istruzione ed *empowerment*. Cosa ha determinato questa capacità di Wikipedia di rimanere una comunità di utenti globalmente responsabile e collaborativa, rispetto al deterioramento della qualità della comunicazione sui social network? La risposta sta nei diversi obiettivi di progettazione: gli algoritmi dei social network sono orientati a massimizzare il tempo di permanenza degli utenti e i guadagni derivanti dalla pubblicità, mentre gli algoritmi di

[15]Crawford 2021; Crawford, Paglen 2019; Harari 2016.

[16]Commissione Europea, Proposal for a regulation of the European Parliament and of the Council laying down harmonised rules on artificial intelligence (artificial intelligence act) and amending certain union legislative acts COM(2021) 206 final , 21 April 2021.

[17]AI Now Institute 2019; Tarnoff 2019.

[18]Mercuri 2021.

Wikipedia servono a monitorare il più possibile i contenuti contro sviste e manipolazioni, senza puntare a crescite economiche di scala. Wikipedia non è l'unico caso di *community-led platform*, cioè piattaforma gestita da persone in modo decentralizzato e collaborativo[19], esistono esempi di applicazioni di *sharing economy* che promuovono comportamenti sostenibili e rinforzano la coesione sociale, soprattutto quando agiscono su scala locale, come *app* di quartiere o cittadine. Le comunicazioni digitali connettono le persone, ma non basta la connessione per creare una comunità: come in tutti i sistemi socio- tecnologici, la differenza dipende dal comportamento e dai valori messi in campo dalle persone, ma anche da **quale tipo di mediazione** è stato scelto come obiettivo di progettazione della piattaforma di connessione[20].

6 Conclusioni

Ad ogni cittadino è oggi richiesto di avere una profonda comprensione di quale sia il ruolo della tecnologia digitale nel determinare il cammino della società, e viceversa di sentire la responsabilità che ha una comunità democratica nel determinare il cammino della tecnologia. Se le scelte non neutrali vengono lasciate all'implementazione dei sistemi socio-tecnologici, allora l'innovazione non porterà vero sviluppo, ma tecnocrazia. Il termine *soluzionismo digitale*[21] indica proprio l'atteggiamento con cui ci si affida a sistemi e applicazioni digitali per risolvere problemi che sono di natura essenzialmente sociale e politica. Ad esempio, una questione complessa come il tracciamento dei contatti delle persone in periodo di pandemia non può essere tradotta nella mera scelta tra un catalogo di soluzioni tecnologiche, così come le scelte politiche non possono schiacciarsi su decisioni prese su dati quantitativi senza comprendere davvero come sono generati quei dati[22].

A questo scopo serve la capacità di guardare ai sistemi digitali con uno sguardo attento ai diversi livelli di astrazione di cui sono composti, distinguendo le criticità che dipendono dai requisiti di progettazione, dalla specifica logica di funzionamento, dalla correttezza dell'implementazione, dall'ambiente di esecuzione e manutenzione, oppure dal contesto di uso. Alcuni di questi aspetti richiedono di possedere maggiori competenze tecniche,

[19] Wagner, B et al. 2021; Bradley, Pargman 2017. 20 Chou 2017.
[20] Chou 2017.
[21] Morozov 2013.
[22] Milan 2020; Pasquale 2019.

ma resta fondamentale la capacità di chiedere conto di come funziona un sistema software, al fine di comprendere come l'informazione è rappresentata nei suoi dati, che manipolazioni vengono effettuate, cosa viene trascurato, chi e come ha fatto le scelte progettuali e implementative, e che garanzie di qualità offre il software. Servono dunque in tutti gli ambiti professionisti capaci di approcci scientifici interdisciplinari, capaci di riconoscere la propria corresponsabilità nell'impatto sociale delle tecnologie digitali. Il Manifesto di Vienna[23] cita anche le Università come luogo di responsabilità per la costruzione del pensiero critico e per l'integrazione del sapere proprio delle scienze umane e sociali con gli studi scientifico-ingegneristici, al fine di orientare il cammino tecnologico verso la promozione della democrazia, l'inclusione e il rispetto dei diritti dell'uomo.

Bibliografia

AI Now Institute. 2019. AI and Climate Change: How they're connected, and what we can do about it, in Medium, https://medium.com/@AINowInstitute/ai-and-climate-change-how-theyre-connected-and-what-we-can- do-about-it- 6aa8d0f5b32c

Bradley, K. - Pargman, D. 2017. The sharing economy as the commons of the 21st century, in Cambridge Journal of Regions, Economy and Society, 10, pp 231–247.

Codenotti, B. - Leoncini M. 2020. La rivoluzione Silenziosa. Le grandi idee dell'informatica alla base dell'era digitale, Codice Edizioni, Torino.

Chou, T. 2017. A Leading Silicon Valley Engineer Explains why Every Tech Worker Needs a Humanities Education, in Quartz. https://qz.com/1016900/tracy-chou-leading-silicon-valley- engi-neer-explains-why-every-tech-worker-needs-a- humanities-education

Crafa, S. – Gaggi, O. 2019. Tecnologia accessibile e societa' inclusiva: binomio possibile?, in Diritti umani e inclusione, Edizioni il Mulino.

[23] Vienna Manifesto on Digital Humanism, 2019 https://dighum.ec.tuwien.ac.at/dighum-manifesto/

Crafa, S – Rizzo, M. 2019. Tecnologie digitali e manipolazione del comportamento. In Mondo Digitale, 86. http://mondodigitale.aicanet.net/2019-7/Articoli/MD86_04_Tecnologie_digitali_e_manipolazione_del_comportamento.pdf

Crawford, K. 2021. Atlas of AI: Power, Politics, and the Planetary Costs of Artificial Intelligence, Yale University Press. Crawford, K. – Paglen, T. 2019. Excavating AI: the politics of images in machine learning training sets. AI Now Institute, NYU. www.excavating.ai

Cutland, N. 1980. Computability. An Introduction to Recursive Function Theory. Cambridge University Press.

Davies, S. 2020. Decoding the social media algorithms in 2020. The ultimate guide. https://www.stedavies.com/social-media-algorithms-guide/

Denning, P .J. - Rosenbloom, P .S. 2009. Computing: The Fourth Great Domain of Science. In Communications of the ACM, 52, 9, pp 27-29.

Harari, Y.N. 2016. Forget about listening to ourselves. In the age of data, algorithms have the answer. Financial Times.

Lewis, P. 2017. Our minds can be hijacked: the tech insiders who fear a smartphone dystopia. The Guardian. https://www.theguardian.com/technology/2017/oct/05/smartp hone-addiction-silicon-valley-dystopia

Menczer, F. - Hills T. 2021. L'economia dell'attenzione. Capire come algoritmi e manipolatori sfruttano le nostre vulnerabilità cognitive ci permette di reagire. Le Scienze.

Mercuri, G. 2021. Wikipedia compie 20 anni: perché è la cosa più fedele allo spirito originario di Internet. Il Corriere della Sera. https://www.corriere.it/tecnologia/21_gennaio_14/wikipedia-compie-20-anni-perche-cosa-piu-fedele-spirito-originario-internet-9d144c7a-5577-11eb-a877-0f4e7aa8047a.shtml

Milan, S. 2020. Techno-solutionism and the standard human in the making of the COVID-19 pandemic, in Big Data & Society,7, 2.

Mitchell M. 2020. Artificial Intelligence. A Guide for Thinking Humans, Pelican Books, Londra.

Morozov, E. 2013. To save everything, click here: the folly of technological solutionism. First edition. New York: Public Affairs.

Nardelli, E. 2019a. Do we really need computational thinking? In Communication of the ACM, 62, 2.

Nardelli, E. 2019b. Informatica e cittadino digitale: dal coding al computational thinking, in Per un'idea di scuola, cap.8, pp.145-166, Lisciani Editore.

Pasquale, F. 2019. Professional Judgment in an Era of Artificial Intelligence and Machine Learning, in Boundary 2, 46(1), pp 73–101.

Pelillo, M. – Scantamburlo, T. 2021 (a cura di). Machines We Trust Perspectives on Dependable AI, MIT Press.

Scantamburlo, T. – Charlesworth, A. – Cristianini, N. 2019. Machine decisions and human consequences. In K. Yeung & M. Lodge (eds) Algorithmic Regulation. Oxford University Press.

Signorelli, A.D. 2019. La gamification di quasi tutto, in Il Tascabile. https://www.iltascabile.com/scienze/gamification-vita- lavoro/

Spode, E.J. 2017. The great cryptocurrency heist. Blockchain enthusiasts crave a world without bankers, lawyers or fat-cat executives. There's just one problem: trust. In Aeon https://aeon.co/essays/trust-the-inside-story-of-the-rise-and- fall-of-ethereum

Tarnoff, B. 2019. To decarbonize we must decomputerize: why we need a Luddite revolution, in The Guardian, https://www.theguardian.com/technology/2019/sep/17/tech- climate-change-luddites-data

Wagner, B et al. 2021. Reimagining content moderation and safeguarding fundamental rights: a study on community-led platforms. Studio commissionato dal gruppo Greens/EFA del Palamento Europeo. https://www.greens-efa.eu/dossier/why- we-need-an-internet-that-works-for-people-and-is-led-by- people/

Wing, J.M. 2006. Computational thinking, in Communications of the ACM, 49, 3, pp. 33-35.