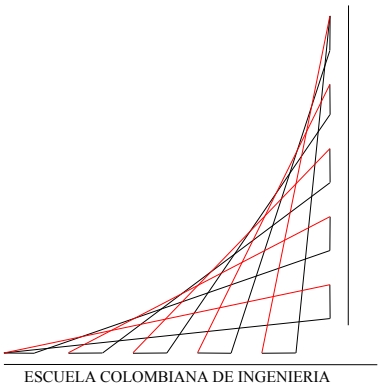


PROYECTO: CURRICULO INTEGRADO PARA LA EDUCACIÓN EN TECNOLOGÍA E INFORMATICA

INFORME FINAL:

LA INFORMÁTICA COMO COMPONENTE CURRICULAR PARA EL CURRÍCULO INTEGRADO EN TECNOLOGÍA E INFORMÁTICA

JAIME A. BOHÓRQUEZ V.
ALFONSO MELENDEZ A.
HERNÁN APONTE M.

ESCUELA COLOMBIANA DE INGENIERÍA
"JULIO GARAVITO"
BOGOTÁ, 14 DE ENERO DE 2003

CONTENIDOS

I. COMPONENTES DEL CURRICULO INTEGRADO EN TECNOLOGÍA E INFORMÁTICA

El "CURRICULO INTEGRADO PARA LA EDUCACIÓN EN TECNOLOGÍA E INFORMATICA" debe orientarse por dos ideas regulativas o ejes referenciales que se articulen orgánicamente, respetando sus diferencias epistemológicas, teórico-metodológicas:

1. El pensamiento crítico sobre los efectos sociales y culturales de la informática y la tecnología así como la creación de ambientes educativos y culturales para aprehenderla (esta orientación es planteada y analizada profusamente en otro documento de este mismo proyecto [D&M02]); y
2. La naturaleza de la informática y la tecnología en general, su evolución, métodos, contenidos y formas de enseñanza apropiados.

En razón de la especificidad disciplinar, en el presente documento nos ocuparemos de una parte del segundo eje referencial: la naturaleza de la informática.

II. ¿QUÉ ES LA INFORMÁTICA?

Cuando todo el mundo conoce la belleza
como belleza, surge la fealdad.
Cuando se conoce el bien como bien,
aparece el mal.
De esta forma, ser y no-ser se producen
mutuamente.

LAO TSE (Tao Te Ying)

Nos narra Platón en el Fedro que cuando el dios Tot inventor de la escritura además de los números, las ciencias y juegos como los dados y el ajedrez, presentó al rey Tamus de Egipto su invención de la escritura, le manifestó: "Oh, rey! este invento hará a los egipcios más sabios y servirá a su memoria; he descubierto un remedio contra la dificultad de aprender y retener". "Ingenioso Tot", respondió el rey, "padre de la escritura y entusiasmado con tu invención, le atribuyes todo lo contrario de sus efectos verdaderos. Ella no producirá sino olvido en las almas de los que la conozcan, haciéndoles despreciar la memoria; fiados en este auxilio extraño abandonarán a caracteres materiales el cuidado de conservar los recuerdos cuyo rastro habrá perdido su espíritu. Cuando vean que pueden aprender muchas cosas sin maestros se tendrán ya por sabios, y no serán más que ignorantes en su mayor parte, y falsos sabios insoportables en el comercio de la vida".

Nos parece interesante imaginar qué contestaría el rey Tamus al dios Tot, si éste se le apareciera con el invento de la informática. Muy seguramente Tot la presentaría también, como un medio de hacer a

los hombres más sabios, como un remedio contra la dificultad de aprender y razonar y un instrumento para imaginar situaciones y realidades alternativas. El rey, a su turno, le contestaría que de nuevo, como ocurrió con la escritura, con el entusiasmo por su invención le atribuye lo opuesto a sus verdaderos efectos. La informática no produciría sino ignorancia de la verdadera razón de las cosas, haciéndoles despreciar el arte del razonamiento; incentivaría además, una incapacidad para imaginar por cuenta propia e incluso, una proclividad a escapar de la realidad y confundirla con el mundo de las percepciones virtuales.

Esta analogía entre la escritura y la informática nos parece pertinente en un doble sentido: por una parte, nos permite establecer un paralelo entre las oportunidades y peligros que nos plantea el efecto revolucionario de la escritura y la informática; y por otra, nos abre puertas hacia la comprensión de la verdadera naturaleza de la informática, la cual permanece oculta cuando solamente se consideran sus manifestaciones externas y su impacto en la sociedad.

Nuestra tesis es que la informática constituye una tercera fase de la evolución del lenguaje humano, precedida por los descubrimientos del lenguaje oral y de la escritura.

Así como la escritura, potenciada por el invento de la imprenta, constituyó una impactante novedad en la historia humana, la informática casi inmediatamente propulsada y encarnada por máquinas electrónicas de computación digital, se nos aparece como una novedad profundamente radical de difícil comprensión y asimilación si nos atenemos a las vertiginosas transformaciones de orden tecnológico, económico y social que ha propiciado.

Es así que la ignorancia, inconsciencia e incomprensión del origen, evolución y naturaleza de la informática ocasionan una "percepción mágica" hacia ella, por parte del común de la gente, que propicia oportunidades de privatización y monopolización de su conocimiento, producción y dominio por grupos de poder e intereses económicos; así como también una cultura de consumo y alienación de características adictivas que confunde conocimiento con abundancia fragmentada de datos e informaciones. De esta manera, se materializan las amenazas y peligros profetizados por el rey Tamus al dios Tot.

La informática, sin embargo, como regalo del dios Tot, constituye una posibilidad real de amplificar y dinamizar los conceptos de texto y escritura mucho más allá de como se conocían previamente. Sus fundamentos hunden sus raíces en los saberes de la matemática, la lógica y la lingüística.

La informática integra aspectos teóricos tanto de la lingüística en lo que se refiere a gramáticas, sintaxis y semántica de lenguajes formales (que incluyen los de programación), como de la lógica matemática en lo que concierne a deducciones y modelos lógicos;

con aspectos prácticos, como los de la ingeniería, la electrónica, y la teoría de la información. En sus fundamentos matemáticos, la informática se distingue de otras ingenierías, como la Civil y la Mecánica, porque estas últimas trabajan en el campo de los fenómenos continuos, mientras que la informática se fundamenta en las matemáticas discretas: la lógica proposicional y de predicados, el álgebra, la teoría de conjuntos, la teoría de grafos, la combinatoria y la probabilidad, entre otras.

Por primera vez en la historia de la humanidad, el potencial humano de la inteligencia igualmente distribuído por el mundo, al tener acceso a la posibilidad de aplicar, desarrollar y crear productos tecnológicos, encuentra la oportunidad de impulsar el desarrollo económico en países menos desarrollados con una mínima infraestructura tecnológica.

El principal requisito para acceder a esta posibilidad es la plena consciencia y conocimiento de su naturaleza, potencialidad y limitaciones, así como también, el reto intelectual y educativo que plantea su apropiación y asimilación tanto ciéntifica y tecnológica como socio-cultural, individual y colectiva.

Una de las tareas indeclinables de un formador integral de la juventud en el área de la tecnología y la informática es la de comunicar esta comprensión y consciencia de la naturaleza real de la informática y de las consecuencias peligrosas y funestas de su “percepción mágica”.

En el numeral V.1 referente a “Fundamentos Teóricos” se presenta una descripción amplia y detallada sobre los fundamentos científicos en que se basa la Informática.

III. EL CONTEXTO

Las fuertes presiones que ejerce la economía global de mercados competitivos y la ansiedad que generan los compromisos nacionales con el ingreso a los mercados libres y con el proceso de globalización internacional han impedido a nuestras sociedades concebir verdaderas estrategias de asimilación y construcción de una cultura científica y tecnológica propias. Es más, en la amplia mayoría de los sectores de la academia, la industria y el mundo comercial dentro del ámbito global -y los nuestros no son la excepción- existe la muy difundida convicción de que la informática como invención, prácticamente está completa y que, en consecuencia, ha "madurado" desde sus inicios como un área teórica de la ciencia hasta llegar a ser un asunto práctico para los ingenieros, los administradores y empresarios, es decir, en su mayoría gente que puede captar la aplicación de la ciencia por sus obvios beneficios, pero que se siente incómoda con su creación porque no entienden la naturaleza de la labor investigativa con sus objetivos intangibles y sus recompensas

inciertas. Esta ampliamente difundida creencia, sinembargo, es solamente correcta si se identifican los objetivos de la informática con lo que se ha logrado y se olvidan aquellos objetivos que no han sido alcanzados, aún siendo demasiado importantes y cruciales para ser ignorados.

Esta creencia resulta propicia para la ideología dominante, ya que contribuye a promover su visión de que la tarea de concebir, diseñar y construir sistemas computacionales corresponde solamente a ciertas compañías multinacionales y centros sofisticados de investigación de países desarrollados pues solamente allí se concentra el capital, el conocimiento y los recursos tecnológicos necesarios y suficientes para producir las "soluciones informáticas" a los problemas de las diferentes sociedades y naciones. A las escuelas, academias y universidades de los países en desarrollo corresponde la "alfabetización tecnológica" de las personas para desempeñarse en los diferentes oficios, artes y profesiones, así como la formación de "profesionales de la informática" capaces de comercializar, gerenciar y adaptar a las empresas e instituciones nacionales, los sistemas informáticos producidos por estas compañías multinacionales, conforme a las idiosincrasias culturales locales y "nativas".

La verdad es que en tanto países en vías de desarrollo no debemos conformarnos con asumir los roles que nos asignan (usuarios y distribuidores de tecnología) pues nuestra capacidad intelectual y operativa nos permite competir en el mundo de la creación de informática. Sobre todo cuando constatamos que la mayoría de los sistemas actualmente en funcionamiento a nivel global, son mucho más complicados de lo que podría considerarse sano, y son demasiado desorganizados y caóticos para ser usados cómoda y confiadamente. El cliente promedio de la industria informática ha sido tan pobremente servido que está habituado a que su sistema se derrumbe todo el tiempo; somos testigos además de una masiva distribución mundial de software plagado de errores que deberían avergonzar profundamente a los miembros de la profesión informática.

No sólo se trata de asignar la responsabilidad por este estado de las cosas a las carencias educativas y formativas del ingeniero promedio, o a la miopía de los administradores o a la ausencia de ética de los distribuidores de equipos y sistemas de computación. Muchos profesionales y académicos alimentan la expectativa ingenua de usuarios comunes y legos en cuanto a que "los computadores asumirán progresivamente cada vez niveles más altos de lo que ahora se considera destrezas, conocimiento e inteligencia humanas"[1]. Aún escuelas contemporáneas de la Psicología al presentar sus

[1] Como bien lo dijo E. W. Dijkstra, la pregunta sobre si una máquina puede pensar, no es más relevante que la pregunta sobre si los submarinos pueden nadar. No deja de ser sorprendente que varios científicos y académicos norteamericanos abriguen este tipo de expectativas. Talvez, como ya lo expresaba Ludwig Wittgenstein en 1940, “la confusión endémica sobre la “gramática” del lenguaje conduce a especulaciones vacías: lejos de ser profunda, la filosofía nos distrae de los asuntos verdaderamente importantes.” [Tou90].

crudos modelos mecánicos como una aproximación válida a la mente humana, han difuminado peligrosamente la diferencia entre el hombre y la máquina. Este tipo de influencias producen la cómoda ilusión de percibir a las personas como sofisticados autómatas a quienes se les puede liberar de la carga de la responsabilidad. La aceptación de la labor de la programación de computadores como un reto intelectual colocaría todo el peso de la responsabilidad sobre sus hombros.

Si se ha de confiar en sus creaciones, la tarea primordial del programador y del ingeniero será diseñar sus artefactos de forma tan comprensible que pueda responsabilizarse por ellos, e independientemente de la respuesta a la pregunta de qué tanto de su actividad actual pueda, en último término, delegarse a las máquinas, debería siempre tenerse en cuenta que ni la "comprensión" ni el "ser responsable" pueden considerarse con propiedad actividades: son más como "estados de la mente" y son intrínsicamente indelegables. Así mismo, la labor de un educador en tecnología informática en la escuela debiera ser la de explicitar y enseñar los medios por los cuales los “métodos simbólicos” provenientes del álgebra y la lógica matemática, dan lugar a métodos de “cálculo lógico” que permiten no sólo resolver incognitas planteads poor ecuaciones, sino realizar raciocinios cuya sencillez, claridad, estructura y corrección pueden ser estudiadas en detalle.

Como lo demuestran múltiples y exitosas experiencias, existe una oportunidad muy general, descrita grosso modo, como la capacidad de diseñar de manera tal que tanto el producto final como el proceso mismo de diseño reflejen una teoría que sea suficiente para prevenir que se cuele una explosión combinatoria de complejidad indómita que haga las cosas inmanejables.

Aunque la oportunidad para abrir campo a la sencillez y la simplificación es muy prometedora, pues en todas las experiencias que vienen a la mente, el proceso de diseño cuesta mucho menos trabajo y conduce mucho mejores productos finales que las alternativas "intuitivamente concebidas". Sinembargo, no se puede esperar tan fácilmente que personas que han dedicado gran parte de su vida profesional a esfuerzos que conducen en la dirección opuesta reaccionen positivamente a propuestas que promulguen la sencillez y la elegancia, por requerir estas cualidades de apropiada educación para apreciarlas así como, de duro trabajo y disciplina para lograrlas. Desgraciadamente la experiencia nos indica que lo que ha resultado suficientemente oneroso es automáticamente declarado un éxito.

IV. ¿QUÉ ENSEÑAR SOBRE INFORMATICA?

Una manera de desmitificar los "aspectos mágicos" de la Informática es conocer y comprender sus conceptos, ideas, métodos y resultados científicos fundamentales. En un sentido muy general, la informática se puede pensar como la ciencia que estudia lo simbólico y lo lingüístico. Desde este punto de vista el computador es una máquina procesadora de símbolos que permite su tratamiento automático: almacenamiento, trasmisión, traducción, combinación, etc.

Aunque inicialmente el computador se consideraba como una máquina calculadora de gran rapidez, la importancia de su impacto en la vida actual se debe fundamentalmente a que los símbolos que manipula ya no solamente representan números y operaciones matemáticas sino entidades y objetos del mundo real. Ha sido tan amplia la gama de situaciones representadas mediante símbolos, que ahora se habla de lo virtual para referirse a toda realidad simulada, representada o recreada en un computador.

A los ojos del profano que comienza a adentrarse en los fundamentos científicos de la informática, y aún a los del experto, resulta sorprendente que en un computador, que dispone solamente de un lenguaje binario -de ceros y unos- se pueda codificar información de tan variada naturaleza y sofisticación. Este poder de codificación, junto con el desarrollo de las telecomunicaciones y la electrónica, explica la introducción del computador en casi todas las áreas de la ciencia y la actividad humana.

Debido a la relativa juventud de esta disciplina científica comparada con la madurez disciplinar de las ingenierías que basan sus aplicaciones en las ciencias naturales, así como también por las percepciones culturales ya descritas, es importante insistir en que en este campo de trabajo continuamente cambiante, muy seguramente, aquellos estudiantes que en su vida profesional lleguen a destacarse serán los que puedan ver más allá de los instrumentos y herramientas del momento y logren armonizarse con el progreso de la disciplina.

Esto no significa que los instrumentos no sean importantes en el oficio. En cualquier área de la ingeniería, las herramientas constituyen una buena parte de la bolsa de elementos que tiene a mano un buen profesional. Debemos sin embargo, mirar más allá de ellas, o través de ellas, hacia los conceptos fundamentales; aquellos que no han cambiado significativamente desde que emergieron, la informática tomó forma hace más o menos 30 años. Al igual que el diseño de las máquinas computadoras (hardware), la tecnología evoluciona, pero los conceptos permanecen. A continuación presentamos los aspectos básicos que reconocidos académicos y profesionales de la comunidad informática mundial (tales como Edsger Dijkstra, David Parnas y Bertrand Meyer) consideran

primordiales para un currículo que equilibre los aspectos científicos ingenieriles y tecnológicos de la informática.

V. ELEMENTOS DEL EJE CURRICULAR

Un currículo debería involucrar cinco elementos complementarios:

1. ***Fundamentos Teóricos***. Los conceptos y bases formales perdurables que hacen posible entender todas las cosas relativas a la ciencia y disciplina informática.
2. ***Principios de Ingeniería Informática***. Conceptos constituidos alrededor del fenómeno de la *complejidad computacional* que corresponden a las cualidades esenciales que debe poseer un sistema informático, y que constituyen la base del campo disciplinar completo y distinguen el quehacer de los profesionales idóneos en esta disciplina.
3. ***Conceptos Prácticos.*** Métodos de solución de problemas que los buenos profesionales de esta disciplina aplican consciente y regularmente.
4. ***Técnicas.*** Prácticas bien establecidas fundamentadas en la experiencia de los maestros reconocidos de esta disciplina.
5. ***Aplicaciones***. Áreas de especialización en las cuales los principios y las prácticas encuentran su mejor expresión.
6. ***Herramientas***. Productos del estado del arte (modernos) que facilitan la aplicación de estos principios y prácticas.

1. Fundamentos Teóricos

La teoría y vocabulario de la informática no aparecieron espontáneamente. Algunos conceptos importantes, tales como los sistemas operativos y los compiladores[2], tuvieron que inventarse "de novo". Otros, tales como la *recurrencia* y la *invariancia*, pueden rastrearse en desarrollos previos de las matemáticas. Llegaron a ser parte permanente del léxico cambiante de la informática, al contribuir con el estímulo o clarificación del diseño y conceptualización de artefactos computacionales. Muchos de estos conceptos teóricos provenientes de diferentes fuentes, están ahora tan inmersos en la computación y las comunicaciones que pululan el pensamiento de todos los científicos de la informática. Muchas de estas nociones, percibidas sólo vagamente en la comunidad informática de la década de 1960, están desde entonces, profundamente imbuídas en la práctica de los profesionales de la informática y han logrado aún llegar en algunos países, a los planes de estudio de la enseñanza media. Aunque los desarrollos de la teoría informática podrían percibirse como intangibles, la teoría subyace a muchos de los aspectos de la construcción, explicación y comprensión de los fenómenos y desarrollos computacionales.

[2] El término apropiado debería ser (sistema) traductor, sin embargo, la historia y la costumbre han hecho del término (sistema) compilador el más usual y aceptado.

En el desarrollo de nuestra comprensión de los fenómenos complejos, la herramienta más potente de que dispone el intelecto humano es la abstracción[3]. La abstracción surge del reconocimiento de las similitudes que existen entre ciertos objetos, situaciones o procesos del mundo real, y de la decisión de concentrarse en esas similitudes, y de ignorar momentáneamente sus diferencias. Tan pronto hemos descubierto qué similitudes tienen importancia para la predicción y control de eventos futuros, tenderemos a considerar las similitudes como fundamentales y las diferencias como triviales. Puede decirse entonces, que hemos desarrollado un concepto abstracto para cubrir al conjunto de objetos o situaciones en cuestión. En ese punto, generalmente introducimos una palabra o un dibujo para simbolizar el concepto abstracto; y cualquier aparición escrita o hablada de la palabra o dibujo, puede utilizarse para *representar* una instancia particular o general de la situación correspondiente.

El propósito primario de las representaciones es transferir a otros información respecto a importantes aspectos del mundo real, y registrar esa información en forma escrita, en parte como una ayuda para la memoria y en parte, también, con el fin de conservarla para las generaciones futuras. Sin embargo, en las sociedades primitivas, se creía a veces que las representaciones tenían una utilidad por sí mismas, puesto que se suponía que la manipulación en sí de las representaciones podía causar los cambios correspondientes en el mundo real; así, sabemos de prácticas tales como clavar alfileres en figurillas de cera que representan enemigos, para causar dolor en la parte correspondiente de la persona real. Este tipo de actividad es característica de la magia y la brujería. Por otra parte, el científico moderno cree que la manipulación de representaciones puede utilizarse para predecir acontecimientos y los resultados de cambios en el mundo real, aunque no para causarlos. Por ejemplo, mediante la manipulación de las representaciones simbólicas de ciertas funciones y ecuaciones podemos predecir la velocidad con que un objeto que cae golpeará la tierra, a pesar de que sabemos que esto no determinará que el objeto caiga ni amortiguará el impacto final cuando así suceda.

La última etapa en el proceso de abstracción es mucho más sofisticada; es el intento de resumir los factores más generales de situaciones y objetos cubiertos por una abstracción, por medio de breves pero poderosos axiomas y demostrar rigurosamente (bajo la condición de que esos axiomas describan correctamente el mundo real) que los resultados obtenidos por la manipulación de representaciones pueden ser aplicados también con éxito al mundo

[3] Resulta inquietante que el documento sobre enfoque crítico social y cultural para el currículo integrado en informática y tecnología dentro de este mismo proyecto [D&M02], considera antagónica esta categoría filosófica tan fundamental para el pensamiento científico moderno en que está basado el desarrollo tecnológico actual. Talvez, la contradicción se explique por "la confusión endémica sobre la "gramática" del lenguaje" de que adolesce la filosofía moderna (y post-moderna?) segun Wittgenstein (ver nota al pie #1), pero esta dilucidación es tarea de filósofos y científicos sociales.

real. Así, los axiomas de la geometría euclidiana corresponden en forma lo suficientemente cercana al mundo real y mensurable como para justificar la aplicación de las construcciones geométricas y de los teoremas a las actividades prácticas de la agrimensura y el relevamiento de la superficie de la tierra.

El proceso de abstracción puede, entonces, resumirse en cuatro etapas:

0) *Abstracción*: la decisión de concentrarse en propiedades que son compartidas por muchos objetos o situaciones del mundo real, e ignorar las diferencias entre los mismos.
1) *Representación*: la elección de un conjunto de símbolos que tomen el lugar de la abstracción; puede usarse como medio de comunicación.
2) *Manipulación*: las reglas para la transformación de las representaciones simbólicas como medio de predecir el efecto de una manipulación similar del mundo real.
3) *Axiomatización*: la enunciación rigurosa de aquellas propiedades que han sido abstraídas del mundo real y que son compartidas tanto por las manipulaciones del mundo real como por las de los símbolos que lo representan.

Como ya lo hemos afirmado, nuestra aproximación a la informática se fundamenta en la convicción de que el proceso de abstracción subyacente en los intentos de aplicar las matemáticas al mundo real, coincide exactamente con el proceso subyacente en las aplicaciones de los computadores al mundo real.

La realización de las etapas de este proceso, ha dado lugar al desarrollo de teorías en los ámbitos:

- lingüístico (etapas de abstracción y representación): la teoría de lenguajes formales
- calculativo o dinámico (etapa de manipulación simbólica): teoría de máquinas abstractas, teoría de calculabilidad, teoría algorítmica, paradigmas de programación, complejidad computacional; criptografía (como teoría de números aplicada)
- algebraico y lógico (etapa de axiomatización): Teoría de corrección de programas, Tipos abstractos de datos

Teorías de carácter lingüístico

Para los circuitos de un computador, y para un programador en código de máquina, cada elemento de datos es una simple colección de bits. Sin embargo, para el programador de JAVA o de C#, un elemento de datos es considerado como un entero, un número real, un vector o una matriz, que son las mismas abstracciones que están subyacentes en las áreas de aplicación numérica para las cuales fueron diseñados, primordialmente, esos lenguajes. Por supuesto, estos conceptos abstractos han sido puestos en correspondencia, por quien implantó el lenguaje con representaciones particulares, mediante conjuntos de bits, en un computador particular dado. Pero

al diseñar su algoritmo, el programador es liberado de la preocupación por esos detalles, que, para sus propósitos, son, en gran medida, irrelevantes; y por lo tanto, su tarea está considerablemente simplificada.

Otra importante ventaja del uso de los lenguajes de programación de alto nivel, como es la independencia respecto de la máquina, también es atribuible al éxito de sus abstracciones. La abstracción puede aplicarse para expresar las características importantes, no sólo de diferentes situaciones reales, sino también de diferentes representaciones de las mismas en el computador. Como resultado, cada implantador puede seleccionar una representación que asegure la máxima eficiencia de manipulación en su computador particular.

Una tercera ventaja importante del uso de un lenguaje de alto nivel es que reduce significativamente los márgenes de error en la programación, Al programar en el código de la máquina, es bastante fácil cometer errores tontos, tales como usar suma con coma fija para números con coma flotante, efectuar operaciones aritméticas sobre índices booleanos. o permitir que direcciones modificadas escapen del rango previsto. Cuando se utiliza un lenguaje de alto nivel, esos errores pueden ser evitados por tres medios:

1) Los errores relacionados con el uso de instrucciones aritméticas equivocadas son imposibles desde un punto de vista lógico: ningún programa expresado, por ejemplo en lenguaje Pascal, puede provocar la generación de códigos de máquina con tales errores.
2) Los errores tales como efectuar operaciones aritméticas sobre variables booleanas serán inmediatamente detectados por el compilador, y nunca afectarán a un programa ejecutable.
3) Los errores tales como el uso de subíndices que exceden el alcance previsto pueden ser detectados por los controles del rango de los subíndices de los arreglos, en el momento de la ejecución.

Los controles durante la ejecución, a pesar de que con frecuencia son necesarios, son casi inevitablemente más costosos y menos convenientes que los controles de los dos tipos anteriores; y los lenguajes de alto nivel deben estar diseñados en forma tal de extender el rango de errores de programación que no puedan ocurrir lógicamente, o que en caso de producirse puedan ser detectados por un compilador. En realidad, un diseño hábil del lenguaje puede permitir que la mayoría de los subíndices sean controlados sin pérdida de eficiencia en la ejecución.

La prevención y detección automática de los errores de programación pueden, asimismo, atribuirse a un uso exitoso de la abstracción. Un lenguaje de programación de alto nivel permite al programador declarar sus intenciones respecto al tipo de los valores de las variables que utiliza, y por lo tanto especificar el significado de las operaciones válidas para los valores de ese tipo. En la actualidad es relativamente sencillo para un compilador verificar la consistencia de un programa y evitar que los errores lleguen a la etapa de ejecución.

El rol de la abstracción en el diseño y desarrollo de programas de computador puede reforzarse utilizando un adecuado lenguaje de programación de alto nivel. En verdad, los beneficios de utilizar un lenguaje de-alto nivel en lugar del código de la máquina pueden deberse, en gran parte, a la incorporación en aquél de abstracciones exitosas, particularmente para los datos.

La *teoría de lenguajes formales* estudia los conceptos y ataca los problemas planteados por estas actividades. El diseño e implantación de lenguajes de programación de alto nivel vía el desarrollo de (sistemas) compiladores e intérpretes requirió del desarrollo de teorías algorítmicas (inspiradas en el concepto de “gramáticas generativas” desarrollado por Noam Chomsky en el contexto de la lingüística de los lenguajes naturales (idiomas humanos)), para realizar mecánicamente *análisis sintácticos* (inglés: *parsing*) de lenguajes formales (como medio de implantar autómatas que interpretaran las acciones prescritas por las directivas expresadas en los programas escritos en lenguajes de alto nivel). A medida que los investigadores fueron comprendiendo la naturaleza del analisis sintáctico, la labor de mecanizar un languaje de programación fue formalizada como tarea de rutina. De hecho, no solamente se automatizó el analisis sintáctico sino también la construcción de analizadores, facilitando de este modo, los primeros de muchos pasos en la conversión del diseño de compiladores de una labor artesanal en una ciencia.

Dadas las profundas diferencias de expresividad, generalidad y manejo del grado de detalle entre los lenguajes usados para describir y modelar el mundo real y las códificaciones por medio de *bits* (unidad binaria con valores cero o uno) de los comandos utilizados para programar instruciones en una máquina, es inevitable tener que trabajar en informática con diversos niveles que permitan pasar de (traducir) conceptos de alto nivel de abstracción lingüística, a instrucciones y datos codificados en lenguaje de máquina, y viceversa.

El proceso de expresión o simbolización (abstracción) de los objetos y eventos del mundo real por medio de un lenguaje lógico (matemático) preciso es llamado en los círculos informáticos *especificación formal*. Estos lenguajes formales modernos tales como *B* y *Z* entre otros, hacen uso de las matemáticas como medio de expresión lingüística, fenómeno éste, que es novel en la práctica usual de las matemáticas.

Otro aspecto de la teoría de lenguajes formales es el estudio de la semántica de los lenguajes de programación. En el caso de los lenguajes de programación, a diferencia de los lenguaje naturales, su semántica (significado) se refiere a la ejecución expresada por el texto de un programa expresado en estos lenguajes. Existen diferentes propuestas de semánticas formales (definidas matemáticamente) para los lenguajes de programación cada con

diferentes usos, propiedades ventajas y desventajas: *semántica operacional* (la ejecución del programa se describe de acuerdo con las operaciones o instrucciones realizadas), *semántica denotacional o funcional* (el programa se describe de acuerdo con un sistema de funciones matemáticas que reflejan los objetivos del programa y que se supone el programa debe evaluar), *semántica axiomática o matemática* (el programa se describe de acuerdo con respecto a los objetivos planteados por su especificación para los cuales la ejecución de su código debe ser correcta).

Teorías de carácter calculativo o dinámico

Los *autómatas* o *maquinas de estados* son modelos ubicuos para describir e implantar los diversos aspectos de la computación. El cuerpo teórico y las técnicas de implantación desarrolladas alrededor del concepto de autómata impulsado la construction y análisis rápidos y precisos de aplicaciones, que incluyen los compiladores, los motores de búsqueda textual, los sistemas operativos, los protocolos de comunicación, y las interfaces gráficas con el usuario.

La idea de un autómata es simple. Un sistema (o subsistema) se caracteriza por un conjunto de estados (o condiciones) que puede asumir. El sistema recibe una serie de entradas (inglés: *inputs*) que pueden hacer que la máquina produzca una salida (inglés: *output*) o pase a un estado diferente, de acuerdo con el estado en que se encuentre. Por ejemplo, un diagrama de estados simplificado de una actividad telefónica podría identificar estados tales como: en espera, tono de marcado, marcando, timbrando, y hablando, así como eventos que ocasionen un desplazamiento de un estado a otro, tales como levantar el auricular, pulsar un dígito, responder, o colgar. Un autómata finito, tal como un telefono, puede estar solamente, en uno de un número limitado de estados. Algunos modelos de autómata más poderosos admiten un número de estados más grande, teóricamente infinito.

La noción de autómata como un modelo de todas las computaciones posibles fue descrito en un muy célebre artículo de Alan Turing sobre la *calculabilidad* en 1936, antes de que los computadores de propósito general se hubiesen inventado. Turing, en la Universidad de Cambridge, propuso un modelo que comprendía una cinta infinitamente larga y un dispositivo que podía leer de, o escribir en, esa cinta [Tur36]. El demostró que tal máquina podía hacer las veces de un computador de propósito general. Tanto en la academia como en la industria, se propusieron y estudiaron modelos relacionados durante las dos décadas siguientes, concluyendo en 1959, con un artículo definitivo de Michael Rabin y Dana Scott de la IBM [Rab59]. En tanto Turing elucidaba la indecidibilidad[4] inherente en el modelo

[4][3] Una clase de problemas matemáticos, usualmente con respuestas si o no, se llama "decidible" si existe un algoritmo que produzca una respuesta definida para cada problema en la clase. De otra forma, la clase de estos problems se llama indecidible. Turing demostró que no existe ningún algoritmo para responder la pregunta sobre si un cálculo en una máquina de Turing terminará o no. La pregunta puede ser respondida por muchas máquinas particulares, aún mecánicamente. pero ningún algoritmo responderá por todas las máquinas: debe haber alguna

más general, Rabin y Scott demonstraron la *tratabilidad* de modelos limitados. Este trabajo habiltó a las máquinas finitas de estados a alcanzar la madurez como un modelo teórico.

Entre tanto, los autómatas y sus equivalentes fueron investigados en conexión con una gran variedad de aplicaciones: redes neuronales [Kle36]; [McC43]; lenguaje [Cho56]; sistemas de comunicación [Sha48], y circuitos digitales [Mea55], [Moo56]. Un nuevo practicabilidad fue demonstrado en un método de derivar eficientemente circuitos secuenciales a partir de autómatas [Huf54].

Cuando los lenguajes formales surgieron como un área de investigación académica en la década de 1960, las máquinas de poder intermedio (es decir, entre autómatas y máquinas de Turing) se convirtieron en tema de investigación. La más notable fue el "autómata de pila" o autómata con una pila (stack) como memoria auxiliar, la cual resulta central para el análisis mecánico usado para interpretar proposiciones (usualmente programas) en lenguajes de alto-nivel.

Complejidad computacional

La teoría de la calculabilidad precedió el advenimiento de los computadores de propósito general y se puede rastrear hasta los trabajos de Turing, Kurt Godel, Alonzo Church, y otros [Dav65]. La teoría de la calculabilidad se concentró en una sola pregunta: ¿Existen procedimientos efectivos para decidir sobre preguntas matemáticas? Los requirimientos del cálculo simbólico han propiciado muchas preguntas detalladas sobre la complejidad intrínseca del cálculo digital, y estas preguntas han estimulado nuevas áreas en las matemáticas.

Los algoritmos concebidos para el cálculo manual se caracterizaban con frecuencia, mediante el conteo de sus operaciones. Por ejemplo, con basa en tales conteos se propusieron varios esquemas para realizar eliminaciones Gaussianas de transformadas finitas de Fourier. Este enfoque se tornó más común con el advenimiento de los computadores, particularmente en conexión con algoritmos de ordenamiento [Fri56]. Sin embargo, el grado inherente de dificultad de los problemas computacionales no llegó a ser un tema de investigación por derecho propio, hasta los años 1960. En la década de 1970, el análisis de algoritmos era ya un tema bien establecido de la informática, y [Knu68] había ya publicado el primer volumen de un tratado sobre el tema que continúa siendo hoy una referencia indispensable. Con el tiempo, los trabajos en teoría de la complejidad han evolucionado tanto como lo han hecho las consideraciones prácticas: de consideraciones concernientes al tiempo necesario para realizar un cálculo, pasando por consideraciones sobre el espacio requerido para ejecutarlo, hasta temas tales como el número de bits

máquina sobre la cual el algoritmo llegará a una conclusión.

aleatorios necesarios para encifrar un mensaje de tal manera que el código no pueda ser fácilmente descifrado.

Al comienzo de la década de 1960, Hao Wang[5] notó distinciones de forma que hacían decidibles algunos problemas de la lógica matemática, en tanto que los problemas lógicos como una clase son indecidibles. Surgió también una sólida clasificación of problemas basados en las capacidades required para que una máquina los pueda atacar. The clasificación fue dramaticamente refinada por Juris Hartmanis y Richard Stearns de la General Electric (GE), quienes mostraron que dentro de un sólo modelo de máquina, existe una jerarquía de clases de complejidad, estratificadas por requerimientos espacio or tiempo. Hartmanis dejó la GE para fundar el departmento de ciencia computación de la Universidad de Cornell. Con el apoyo de la National Science Foundation (NSF), Hartmanis continuó el estudio de la complejidad computacional.

Hartmanis y Stearns desarrollaron un teorema de "aceleración", que en esencia decía que la jerarquía de complejidad no se ve afectada por la velocidad de computación subyacente. Lo que distingue los niveles de la jerarquía es la manera en que el tiempo de solución varía con el tamaño del problema --y no con la escala con la cual se mide el tiempo. En consecuencia, es útil hablar de complejidad en términos of orden-de-crecimiento. Para este fin, la notación "o mayúscula", de la forma O(n), fue importada del análisis de algoritmos a la computación (muy notablemente por [Knu76]), donde ha tomado vida propia. La notación usada para describir la tasa a la cual el tiempo necesario para generar una solución varía con el tamaño del problema. Los problemas para los cuales hay una relación lineal entre el tamaño del problema y el tiempo para obtener una solución son O(n); aquellos para los cuales el tiempo para obtener una solución varía como el cuadrado del tamaño del problema son $O(n^2)$.[6] Las estimaciones “o mayúscula” muy pronto pulularon los cursos de algorítmica y desde entonces, han sido incluídos en los planes de estudio en informática de la enseñanza media.

El enfoque cuantitativo de la complejidad iniciado por Hartmanis y Stearns se difundió rápidamente en la comunidad académica. Aplicando este agudo punto de vista a problemas de decision de la lógica, Stephen Cook en la Universidad de Toronto propuso la más célebre noción teórica de la informática—la completitud NP. Su conjetura "P versus NP" se cuenta ahora entre los problemas abiertos más importantes de las matemáticas. Afirma que existe una muy marcada distinción entre los problemas que pueden ser calculados determinadamente o no-determinadamente en una cantidad tratable

[5] Hao Wang, quien inició su trabajo en la Universidad de Oxford y posteriormente se mudó a Bell Laboratories y la IBM, elucidó las fuentes de la indecidibilidad.

[6] Como ejamplo de la notación “o mayúscula”, el número de objetos sólidos idénticos en tamaño que pueden caber en un cubo con lados de longitud L es $O(L^3)$, independientemente del tamaño o forma de los objetos. Esto significa que para L arbitrariamente grande, a lo sumo cabrán L^3 objetos (escalados por alguna constante).

de tiempo.[7] La teoría de Cook, y el trabajo previo de Hartmanis y Stearns, ayudó a categorizar los problemas como determinados or no-determinados. La importance práctica del trabajo de Cook fue vivificada por Richard Karp, de la Universidad de California en Berkeley (UC-Berkeley), quien demostró que una collección of problemas tratables no-determinadamente, que incluía al famoso problema del agente viajero,[8] eran intercambiables ("NP complete") en el sentido de que, si alguno de ellos es determinadamente tratable, entonces todos ellos lo son. Un torrente de otros problemas NP-completos aparecieron a continuación, desencadenado por un libro seminal de Michael Garey y David Johnson de los Laboratorios Bell [Gar79].

La conjetura de Cook, si es verdad, implica que no hay esperanza de solucionar con precisión cualquiera de estos problemas en un computador real sin pasar por la pena de durar tiempo exponencial. Como consecuencia, los diseñadores, de sistemas informáticos sabiendo que ciertas aplicaciones (e.g., integrated-circuit layout) son intrínsecamente difíciles, pueden optar por soluciones "suficientemente buenas", en vez de buscar las "mejores posibles". Esto conduce a otra pregunta: Qué tan buenas soluciones puede obtenerse con una cantidad de esfuerzo dada? Una teoría más refinada sobre soluciones aproximadas para problemas difíciles ha sido desarrollada [Hoc97]. Afortunadamente, existen buenos métodos de aproximación para algunos problemas NP-completos. Por ejemplo, gigantescas "rutas para agentes viajeros" se utilizan rutinariamente para minimizar la trayectoria de un taladro automatizado sobre el plano de un circuito para el cual miles de orificios deben perforarse. Estos métodos de aproximación son suficientemente buenos para garantizar que ciertas soluciones fáciles aparecerán muy cercanas a (es decir., dentro de un 1 por ciento de) la mejor solución posible.

[7] Una clase de problemas se dice "tratable" cuando el tiempo necesario para resolver problemas de la clase varía a lo sumo, como una potencia del tamaño del problema.

[8] Este problema tiene que ver con encontrar la ruta más eficiente que debe seguir un agente viajero para visitar una lista de ciudades. Cada ciudad adicional que se añada a la lista crea una serie completa de posibles rutas adicionales que deben ser evaluadas para identificar la más corta. Por tanto, la complejidad del problema crece mucho más rápido que lo que lo hace la lista de ciudades.

Teorías de carácter algebraico y lógico

El primer requisito al diseñar un programa, es concentrarse en las características relevantes de la situación, e ignorar los factores que se consideran irrelevantes. Por ejemplo, al analizar las características vibroelásticas de un diseño de ala propuesto para una aeronave, su elasticidad es lo que se considera relevante: su color, forma, técnica de producción, son considerados irrelevantes excepto en la medida en que han contribuido a su elasticidad. Tomando un ejemplo comercial, los empleados que trabajan para una compañía tienen muchas características, tanto físicas como mentales, que serán ignoradas al diseñar un sistema de sueldos para dicha compañía.

La siguiente etapa en el diseño del programa, es la decisión respecto a la forma en que la información abstraída será representada en el computador. Una función de elasticidad puede ser representada por sus valores en un número apropiado de puntos discretos; y éstos pueden ser representados en una variedad de formas como un arreglo de dos dimensiones. Alternativamente, la elasticidad puede estar dada por una función calculada y los datos ser mantenidos como un vector de coeficientes polinomiales o de Chebychev para la función. Un archivo de sueldos en un computador consta de cierto número de registros, cada uno relacionado con un empleado. La elección de la representación que tendrá dentro del registro cada atributo relevante, debe ser adoptada como parte del diseño del programa.

La etapa de la axiomatización no es considerada usualmente como una etapa separada en la programación; y, frecuentemente, se la deja implícita. En el caso de la vibroelasticidad de una aeronave, la axiomatización es la fórmulación de las ecuaciones diferenciales que se supone describen la reacción del ala real a cierto tipo de tensiones, y que (se espera) describen también el proceso de solución aproximada en el computador. En el caso de la planilla de sueldos, los axiomas corresponden a las descripciones de distintos aspectos del mundo real que deben ser incluidos en el programa; por ejemplo, el hecho de que el pago neto es igual al pago bruto menos las deducciones.

Finalmente, se llega a la tarea de programar el computador para que efectúe esas manipulaciones de las representaciones de los datos, que corresponden a las manipulaciones, en el mundo real, en que estamos interesados. El éxito de un programa, depende de tres condiciones básicas:
1) La axiomatización es una descripción correcta de aquellos aspectos del mundo real con los que está relacionada.
2) La axiomatización es una descripción del comportamiento del programa; es decir, que el programa no contiene errores.
3) La elección de la representación y el método de manipulación son tales que el costo de procesar el programa en el computador sea aceptable.

Para simplificar la tarea de diseño y desarrollo de un programa de computador, es muy útil poder mantener razonablemente separadas estas tres etapas y cumplirlas en la secuencia apropiada. Así, la primera etapa (axiomatización) debería culminar con una enunciación rigurosamente lógica de las suposiciones acerca del mundo real, y una formulación de los objetivos deseados, que deben ser alcanzados por el programa.

La segunda etapa debería culminar con un algoritmo, o programa abstracto, que sea demostrablemente capaz de desarrollar la tarea planteada, sobre la base de los supuestos dados. La tercera etapa debería ser la decisión respecto a cómo estarán representados y cómo serán manipulados los distintos elementos de datos almacenados en la computadora, a fin de lograr una eficiencia aceptable. Sólo cuando esas tres etapas hayan sido satisfactoriamente concluidas, debería comenzar la fase final de codificación y prueba del programa, que es la corporización de la operación del algoritmo elegido, sobre la representación de datos elegida.

Por supuesto, esta es una imagen algo idealizada de la tarea intelectual de programar, como una nueva progresión desde la formulación abstracta del programa, a los aspectos más y más concretos de su solución. En la práctica aún en la formulación de un problema, el programador debe tener alguna intuición respecto a la posibilidad de una solución; mientras diseña su programa abstracto, debe tener una cierta sensación de que dispone de una representación adecuadamente eficiente. Con mucha frecuencia, esas intuiciones y sensaciones serán erradas, y una investigación más profunda de la representación, o inclusive la codificación final, demandará una vuelta a una etapa anterior del proceso y quizás, inclusive, un replanteo radical de la dirección de ataque. Pero este ejercicio de previsión intuitiva, junto con el riesgo de fracaso, es característico de todos los procesos intelectuales constructivos e inventivos, y no reduce los méritos de comenzar, por lo menos, en forma ordenada, con etapas más o menos claramente separadas. Una de las características más importantes de la progresión es que la codificación real del programa ha sido pospuesta hasta después de que sea (casi) seguro que todos los otros aspectos del diseño han sido completados con éxito. Puesto que la decodificación y prueba de un programa es, en general, la etapa más costosa en el desarrollo del mismo; no es deseable tener que efectuar cambios después de haber comenzado esta etapa. Es, por lo tanto, ventajoso asegurarse de antemano que ningún otro elemento falle en esta etapa final; por ejemplo, que el programa resuelve el problema correcto, que el algoritmo es correcto, que las distintas partes del programa cooperan armoniosamente en la tarea total, y que las representaciones de datos son adecuadamente eficientes.

Teorías de corrección[9] de programas

Aunque los primeros algoritmos de computador fueron escritos principalmente para resolver problemas matemáticos, sólo existía una tenue e informal conexión entre los programas de computador y las ideas matemáticas que ellos pretendían implantar. La brecha entre los programas y las matemáticas se amplió con el surgimiento de la programación de sistemas, la cual se concentraba en los mecanismos de interacción con un ambiente de computación en vez de las matemáticas.

La posibilidad de tratar el comportamiento de los programas como tema de un argumento matemático fue adelantada de manera vigorosa por Robert Floyd en la Universidad de California en Berkeley, y más tarde amplificada por Anthony Hoare en la Universidad de Queen en Belfast. El movimiento académico hacia la verificación de programas fue acompañada por un movimiento hacia la programación estructurada, definida así por Edsger Dijkstra en la Universidad Técnica de Eindhoven y vigorosamente promovida por Harlan Mills en la IBM y muchos otros. Un supuesto básico de este último movimiento era que la buena estructuración de los programas favorece la habilidad para razonar sobre ellos y por tanto asegurar su corrección. Más aún, una estructuración análoga era conformar el proceso de diseño mismo, conduciendo a una más alta productividad así como a mejores productos. La programación estructurada llegó a ser un lema obligatorio en los textos de programación y una práctica obligatoria en muchas firmas importantes de software.

En el enfoque pleno hacia la derivación formal de programas, las especificaciones de un programa son descritas matemáticamente, y se deriva una prueba formal de que el programa cumple con dichas especificaciones. Para asegurar la validez de pruebas (tediosamente largas) éstas se realizarían o revisarían mecánicamente. Hasta la fecha, este enfoque ha sido demasiado oneroso para ser contemplado por la práctica cotidiana de la programación. Sin embargo, los simpatizantes de la programación estructurada promueven algunas de sus ideas claves: precondición, postcondición e invariantes. Estos términos han llegado hasta los currículos de ciencias de computación, aún al nivel de enseñanza media. Sea o no que la lógica sea explícitamente enunciada en el código escrito por los programadores cotidianos, estas ideas son parte esencial de su trabajo.

La perspectiva de la programación estructurada condujo a una disciplina más avanzada, promulgada por David Gries en la Universidad de Cornell y Edsger Dijkstra en Eindhoven, la cual comienza a hacer parte de los currículos. En este enfoque los programas son derivados a partir de sus especificaciones por cálculos algebraicos. En las manifestaciones más avanzadas formuladas por

[9] La corrección se define como la propiedad de ser consistente con una especificación.

Eric Hehner, la programación se identifica con la lógica matemática. Aunque queda por ver si el grado de matematización eventualmente llegara a ser práctica común, la historia del análisis ingenieril sugiere que este evento es muy probable.

En un área, el diseño de sistemas distribuídos, la matematización, se difunde en el campo de aplicación quizás más rápido que en el aula de clase. El impetu inicial fue la validación de West de un protocolo estándar propuesto internacionalmente. El tema rápidamente maduró tanto en la práctica [Hol91] como en la teoría [V&W86]. Hoy en día, los ingenieros dominan una plétora de álgebras (tales como la lógica temporal, el álgebra de procesos) en herramientas prácticas para analizar protocolos utilizados en aplicaciones que van desde los buses de hardware hasta las comunicaciones por Internet.

Es particularmente difícil prever los efectos de eventos anormales en el comportamiento de las aplicaciones en comunicaciones. La pérdida o corrupción de mensajes entre computadores, o conflictos entre eventos concurrentes tales como agentes viajeros inscribiendo la misma silla en un mismo vuelo de una aerolínea, pueden causar inconvenientes o aún catástrofes como fue notado por [Neu95]. Estas dificultades de la vida real han estimulado la investigación en análisis de protocolos lo que hace posible predecir comportamientos bajo un rango completo de condiciones y eventos, y no simplemente unos pocos escenarios simples. Un cuerpo teórico y una práctica han emergido en la década pasada para hacer del análisis automático de protocolos una realidad práctica.

Teoría algebraica de tipos abstractos de datos

La teoría de estructuración de datos depende fuertemente del concepto de tipo. Este concepto es familiar a los matemáticos, lógicos y programadores.

1) En el razonamiento matemático, es usual establecer una distinción más bien marcada entre individuos, conjuntos de individuos, familias de conjuntos, etcétera; establecer distinciones entre funciones reales, funciones complejas, funcionales, conjuntos de funciones, etc. En realidad, para cada nueva variable que introduce en su razonamiento, el matemático declara inmediatamente qué tipo de objeto representa la variable, por ejemplo:

"Sea *f* una función real de dos variables reales"
"Sea *S* una familia de conjuntos de enteros"

A veces en un texto matemático se da una regla general que relaciona el tipo de símbolo con ciertos tipos particulares de impresión, por ejemplo:

"Utilizamos letra redonda minúscula para representar individuos, mayúsculas para representar conjuntos de individuos, y mayúsculas de tipo inglés para denotar familias de conjuntos."

En general, los matemáticos no utilizan convenciones de este tipo para hacer distinciones arbitrarias; por ejemplo en general no se las utilizarían para distinguir números primos de no primos, ni grupos abelianos de grupos generales. En la práctica, las convenciones de tipo adoptadas por los matemáticos son muy similares a aquellas que serían de interés para los lógicos y programadores.

2) Los lógicos, en general, prefieren trabajar sin variables tipificadas. Sin embargo, sin tipos, dentro de la teoría de los conjuntos, es posible formular ciertas paradojas que llevarían a una contradicción ineludible y al derrumbe del razonamiento lógico y matemático. La más famosa de éstas es la paradoja de Russell:

"Sea *s* el conjunto de todos los conjuntos que *no* son miembros de si mismos. ¿Es *s* un miembro de sí mismo o no?"

Y resulta que ya se conteste sí o no, puede demostrarse inmediatamente que la respuesta es errada.

La solución de Russell a la paradoja, es asociar a cada variable lógica o matemática un *tipo,* que define si es un individuo, un conjunto, un conjunto de conjuntos, etc. Luego establece que cualquier proposición de la forma "*x* es un miembro de *y*" tiene gramaticalmente sentido sólo si *x* es una variable de tipo individual e *y* es del tipo conjunto, o si *x* es del tipo conjunto e *y* es del tipo conjunto de conjuntos, etcétera. Cualquier proposición que viole esta regla es considerada sin sentido, la cuestión de su verdad o falsedad simplemente no se plantea, es solamente una mezcla de letras. Así, cualquier proposición que implique conjuntos que son o no son miembros de sí mismos, puede, sencillamente, ser dejada de lado.

La teoría de los tipos de Russell lleva a ciertas complejidades en los fundamentos de las matemáticas, que no es pertinente describir aquí. Lo interesante para nuestros propósitos es que los tipos permiten evitar que se utilicen ciertas expresiones erróneas en las fórmulas lógicas y matemáticas, y que una verificación de que no se violan las restricciones de los tipos puede efectuarse simplemente examinando el texto, sin conocimiento del valor que pueda llegar a tener un símbolo en particular.

3) En un lenguaje de programación de alto nivel, el concepto de tipo es de importancia fundamental. Nuevamente, cada variable, constante y expresión tiene asociado un único tipo. En PASCAL y ALGOL la asociación de un tipo con una variable se establece mediante su declaración; en FORTRAN, se la deduce de la letra inicial de la variable. En la implantación del lenguaje, la información de tipo determina la representación de los valores de la variable, y la cantidad de almacenamiento en el computador que debe serle asignado. La información respecto al tipo también determina la forma en que deben ser interpretados los operadores aritméticos; y permite al compilador rechazar por faltos de sentido a aquellos programas que suscitan operaciones inapropiadas.

Así, existe un alto grado de concordancia en el uso del concepto de tipo por parte de matemáticos, lógicos y programadores. Las características salientes del concepto tipo, pueden ser resumidas:

1) Un tipo determina la clase de valores que puede asumir una variable o expresión.
2) Cada valor pertenece a un tipo y sólo a uno.
3) El tipo de un valor denotado por cualquier constante, variable o expresión, puede deducirse de su forma o contexto, sin tener conocimientos sobre su valor, calculado en el momento de la ejecución.
4) Cada operador espera operandos de algún tipo fijo, y entrega el resultado en algún tipo fijo (generalmente el mismo). Cuando el mismo símbolo es aplicado para varios tipos distintos (por ejemplo, + para la suma de enteros y las de reales), ese símbolo puede ser considerado ambiguo, denotando varios operadores concretos diferentes. La resolución de esa ambigüedad sistemática. siempre puede hacerse en el momento de la compilación.
5) Las propiedades de los valores de un tipo y de las operaciones primitivas definidas respecto de los mismos, son especificadas por medio de un conjunto de axiomas.
6) En un lenguaje de alto nivel la información sobre el tipo es utilizada tanto para evitar o detectar construcciones sin sentido en un programa, como para determinar el método de representación y manipulación de datos en una computadora.
7) Los tipos en que estamos interesados, son aquellos ya familiares al matemático; es decir, los productos cartesianos, uniones discriminadas, conjuntos, funciones, sucesiones y estructuras recursivas.

Conclusión

La informática, como tercera fase de la evolución del lenguaje humano, en lo que a sus aspectos lingüísticos se refiere, se fundamenta teóricamente en ciencias exactas o formales relacionadas con la lingüística y la lógica matemática. Tanto su naturaleza como su desarrollo y aplicación requieren apelar a conceptos, raciocinios y maneras de hacer todavía incipientes (datan de hace apenas treinta años), aún haciendo acopio del propio quehacer y bagaje de conocimientos que proporcionan las ciencias matemáticas. Se vislumbra que este mutuo aporte entre la informática (como ciencia lógico-lingüïstica) y las ciencias matemáticas será tan fructífero y enriquecedor como el que se dió entre la física (como ciencia natural) y las matemáticas.

A nuestro juicio un conocimiento y comprensión básicos de los fundamentos teóricos de la informática constituye el antídoto principal contra su “percepción mágica”. Esta posición se traduce en que contrariamente a la creencia generalizada, la comprensión profunda de la informática no depende ni única ni esencialmente del dominio técnico-operativo de las máquinas (hardware) y los sistemas informáticos (software).

La diferencia en calidad y énfasis con respecto a la manera tradicional en que operan y se han desarrollado las matemáticas como ciencia, estriba en los colosales grados de tamaño y detalle que se requieren para construir sistemas informáticos crecientemente más y más complejos. Los fenómenos negativos que (frecuentemente) se producen cuando estos requerimientos de tamaño y detalle no son adecuadamente manejados son denotados genéricamente con el término de *complejidad*.

La habilidad de razonar formalmente es una habilidad muy valorada en un programador. Los estudiantes que han dominado la habilidad de aplicar razonamientos matemáticos al desarrollo informático tienen una gran ventaja sobre aquellos que no. Para ser un profesional idóneo se requiere haber dominado algunos aspectos matemáticos de la informática que son extremadamente útiles para modelar problemas de software en términos formales.

Esta habilidad debe ser parte de un currículo universitario fuerte, y debe ser enseñada en los primeras fases de la carrera, antes de que los estudiantes hayan tenido contacto con medios de producción empresariales o industriales.

De otra parte, las habilidades y conocimientos matemáticos que son relevantes para un ingeniero de sistemas son distintos a los de otras ingenierías. Tal vez baste con estudios no muy profundos de cálculo, lógica matemática, álgebra, matemáticas discretas (conjuntos, relaciones, funciones, probabilidad, grafos, entre otros), teoría de lenguajes, y alguna práctica con lenguajes formales modernos útiles para expresar especificaciones y diseños de sistemas informáticos tales como Z y B.

2. Principios de la Ingeniería Informática

Los principios de la ingeniería informática se han ido constituyendo alrededor del concepto de complejidad (explicado anteriormente) como requerimientos ineludibles de la calidad de los sistemas informáticos. Las cualidades que a continuación se describen, surgen naturalmente al intentar enfrentar las dificultades teóricas y tecnológicas que se presentan al enfrentar los factores de tamaño y detalle en muy grandes proporciones, que constituyen el fenómeno de *complejidad computacional*.

Un concepto que ha demostrado ser de gran utilidad para lidiar y eludir los fenómenos de complejidad es el de *módulo* o *componente* (informático)[10]. Este concepto ha ido evolucionando en el tiempo a medida que se hace más preciso, adquiere cualidades específicas y da lugar a formas y estrategias de concebir y diseñar sistemas

[10] No existe definición única para este concepto, sino que más bien, ha ido evolucionando con el tiempo. Talvez, los conceptos de "ocultamiento de información" y de "clase" (implantación de un tipo abstracto de datos) proveniente de la "programación orientada por objetos", han sido la mayor contribución al perfeccionamiento de este elusivo concepto.

informáticos que se conocen con el nombre de *arquitectura de diseño*.

El cuerpo teórico de la ciencia informática se inspira en estudiar la naturaleza y fenómenos que conllevan estas cualidades, así como los problemas que plantea la construcción de sistemas informáticos que las posean. Algunos de los principios de la informática como actividad ingenieril son:

Corrección: la habilidad de un sistema informático para desempeñar sus tareas con exactitud, de acuerdo con la manera en que se definen en sus especificaciones.

La corrección es la cualidad primordial de un sistema informático. Si un sistema no hace lo que se supone que debe hacer, todo lo demás, —que sea rápido, fácil de usar, funcional, . . .— importa poco.

Sinembargo, la comprensión y el logro de esta cualidad ha planteado un reto fundamental a la ciencia informática. Aún el primer paso hacia la corrección es ya bastante difícil: la *especificación* precisa de los *requerimientos* de un sistema informático plantea retos de orden lingüístico a la lógica formal, desconocidos en la práctica tradicional de las matemáticas.

La ausencia de orden y método, así como la ineficacia de las técnicas de “comprobación a posteriori” (testing) y de “depuración de fallas” (debugging) para garantizar la *correción de los sistemas informáticos con respecto a su especificación*, dió lugar a un “estado de conmoción interior” dentro de la comunidad informática mundial, conocida por sus miembros como la *crisis del software*, que fue declarada oficialmente en 1968 y hasta el día de hoy no ha sido levantada.

Uno de los logros teóricos fundamentales de la informática en los últimos treinta años ha sido la dilucidación y completa caracterización matemática del concepto de corrección con respecto a una especificación, por lo menos para la “programación en pequeña escala”.

La “programación a gran escala” como ya se ha señalado, plantea por sí misma problemas adicionales, que seguramente requerirán de más y más profundos resultados teóricos. Los métodos para asegurar corrección a gran escala son usualmente condicionales, se hacen necesarios *enfoques estratificados* donde la garantía de corrección de las propiedades y componentes de un nivel o estrato presupone la corrección de los niveles inferiores.

Solidez: la habilidad de un sistema informático para reaccionar apropiadamente a condiciones anormales.

La solidez es complementaria a la corrección. La corrección tiene que ver con el comportamiento de un sistema en los casos cubiertos por su especificación, la solidez caracteriza el comportamiento por fuera de dicha especificación.

Eficiencia: habilidad de un sistema informático para requerir tan pocos recursos computacionales como sea posible, tales como tiempo de procesamiento, espacio ocupado en almacenamiento interno y externo (a la máquina procesadora), ancho de banda utilizado en dispositivos de comunicación.

Aunque en un segundo plano con respecto a la corrección, en algunos casos la eficiencia puede afectarla: la especificación de un sistema puede requerir que la respuesta del computador a un cierto evento no ocurra después de a cierto lapso predeterminado.

El concepto de eficiencia ha motivado el desarrollo de teorías matemáticas de complejidad computacional temporal y espacial de algoritmos.

Extensibilidad: facilidad de un sistema informático para adaptarse a cambios en su especificación.

En principio se supondría que los sistemas informáticos son muy flexibles (software), nada puede ser más fácil que cambiar un programa, si se tiene acceso a su código. Simplemente use su procesador de texto favorito.

El problema de extensibilidad es de escala. Para programas pequeños, el cambio no es un asunto de consideración; pero a medida que un sistema informático crece, se vuelve más y más difícil de adaptar. Con frecuencia, un gran sistema informático es percibido por el personal encargado de su mantenimiento, como un castillo de naipes gigantesco, en el que el retiro o modificación de uno de sus elementos puede ocasionar que la estructura entera se derrumbe.

El cambio en el desarrollo de sistemas informáticos es ubicuo y permanente: cambio en los requerimientos, de la comprensión de los requerimientos, de los algoritmos, de la representación de los datos, de las técnicas de implantación. Los siguientes dos principios subordinados son esenciales para obtener progresos con la extensibilidad:

- *Sencillez en el diseño:* Una *arquitectura* simple es siempre más fácil de adaptar a cambios que una compleja.
- *Descentralización*: Entre más autónomos los *módulos* informáticos más grandes las posibilidades de que un cambio simple afecte sólo a un módulo, o a un número pequeño de módulos, en lugar de desatar una reacción en cadena de cambios sobre todo el sistema.
- *Reutilización*: la habilidad de los elementos informáticos (constituyentes de un sistema) para permitir la construcción de muchas otras aplicaciones diferentes.

Los ingenieros de sistemas actuales no pueden estar reinventando la rueda cada vez que realizan un proyecto, y una decisión usual y clave que deben tomar con frecuencia es cuándo confiar en el trabajo desarrollado por alguien, y hacerlo parte del suyo. Reutilizar bien es

una habilidad; producir resultados para que otros reutilicen fácil y satisfactoriamente es característica de profesionales.

Facilidad de uso: Facilidad con la cual personas de diversos contextos culturales y calificaciones pueden aprender a usar sistemas informáticos y aplicarlos en la solución de sus problemas. Cubre también la facilidad de instalación operación y monitoreo.

Como con muchas de las otras cualidades relacionadas, una de las claves que un sistema sea fácil de usar es la sencillez estructural. Un sistema bien diseñado, construido de acuerdo con una estructura clara y bien concebida tenderá a ser más fácil de aprender y usar que una compleja y confusa. Esta condición por supuesto, no es suficiente pero es de gran utilidad.

Principios o cualidades adicionales

Existen muchas otras cualidades adicionales que afectan a los usuarios de los sistemas informáticos, mencionamos algunos más para dar una idea de la abrumadora complejidad a la que se ve abocado quien emprende el desarrollo de un sistema informático de tamaño considerable.

Integridad Es la habilidad de un sistema informático para proteger sus varios componentes (programas, datos) contra accesos y modificaciones no autorizados.

Funcionalidad: amplitud de posibilidades ofrecidas por un sistema informático.

Oportunidad: habilidad de un sistema informático de estar disponible cuando o antes de que el usuario lo requiera.

Compatibilidad: facilidad de combinar un elemento informático con otro.

Portabilidad: facilidad de trasladar sistemas informáticos a diversos ambientes computacionales (hardware) e informáticos (software).

2. Conceptos Prácticos

Entre las cosas más importantes que un profesional de la informática debe conocer están los conceptos que son recurrentes en todo el desarrollo su trabajo. La mayoría de estos conceptos no son métodos específicos. Si éstos involucran un método, van más allá de él para dar lugar a un modo de razonamiento. Esto define el aspecto más emocionante de ser un profesional de la informática: El dominio de algunos de estos esquemas intelectuales, poderosos y elegantes, que más que cualquier truco particular del oficio, constituyen el tesoro común de los profesionales de la informática. La mayoría de ellos no pueden ser enseñados en una sola sesión, ni tal vez en un solo curso, sino que se aprenden poco a poco mediante ensayo, error y guía de un experto.

Al igual que los fundamentos teóricos y los principios, los conceptos forman el más importante cuerpo de conocimiento que un profesor debe trasmitir a sus estudiantes, no solamente mediante palabras; es un conjunto de conceptos creativos que vienen poco a poco de la misma manera en la que un programador aficionado progresa hasta convertirse en un maestro del oficio. El dominio de estos principios y conceptos es lo que tal vez nos autorice a considerarnos profesionales.

Abstracción. Esta es la herramienta intelectual clave –la habilidad para separar lo esencial de lo secundario, para ver la idea fundamental por encima de su realizaciones particulares. Muchísimo más que encualquier otra disciplina de la ingeniería, la abstracción es un arma fundamental para combatir la complejidad.

Distinción entre especificación e implantación. Conseguir esta apropiada distinción es un tema constante en las discusiones en ingeniería de software. Es un problema propio de la informática, porque tratamos con cantidades o entidades etéreas y virtuales. Nadie confundiría un puente con el dibujo del puente o un carro con el plano del carro. Pero en software, la distinción está lejos de ser clara.

Recurrencia. Se relaciona con las definiciones que se apoyan en el concepto que definen sin por ello incurrir en un círculo vicioso. El problema no son solamente las rutinas recurrentes –una técnica– sino el modo general de razonamiento que define un concepto por aplicación de la definición misma a algunas de sus partes. Es algo confusa al principio, pero una vez se ha aprendido a usar la recurrencia apropiadamente, se ha ganado una herramienta intelectual poderosa, aplicable a lo largo de todo el campo disciplinar.

Ocultamiento de información. Una poderosa estrategia para combatir la complejidad es la de que los módulos compartan de manera explícita (mediante un mecanismo de interfaz), únicamente la información indispensable toda otra información debe ser privada al módulo al cual pertenece y por tanto inaccesible desde otros módulos. Decidir qué información se exporta al mundo exterior y qué información se conserva privada dentro de un módulo de un sistema informático es una habilidad que los desarrolladores informáticos aprenden únicamente a través de una combinación de buenos ejemplos y prácticas permanentes.

Combatir la complejidad. Como se ha reiterado, los sistemas informáticos constituyen empresas intelectuales ambiciosas; y la complejidad computacional amenaza con abrumar a sus constructores a cada paso. Los expertos saben como reconocer la sencillez esencial detrás de un enredo aparente. Lo que se pide construir es cada día más complejo y de mayor tamaño; por esto mismo la presión se centra cada vez más, en desarrollar técnicas y herramientas de todo estilo que nos permitan lidiar adecuadamente con la complejidad.

Consciencia de la escala. La complejidad es un problema de escala. Algunos sistemas informáticos que se producen no son solamente complejos, su mero tamaño puede ser asombroso. El código fuente de Windows 2000 tiene alrededor de 35 millones de líneas de código de programación y algunos sistemas de defensa o telecomunicaciones están en el mismo rango. Unos cuantos millones de líneas son hoy en día muy comunes. Muchos problemas de la ingeniería informática aparecen cuando el tamaño crece; la cantidad afecta la calidad. Parte de las habilidades de un buen profesional es saber cuáles técnicas son escalables y cuáles no.

Diseño evolutivo para el cambio constante. Los sistemas informáticos cambian más que cualquier artefacto de ingeniería de otro tipo. A menos que los desarrolladores apliquen minuciosamente principios arquitectónicos estrictos, el proceso de cambio puede ser penoso, especialmente para sistemas grandes. Gran parte de la justificación para los modernos métodos, lenguajes, y herramientas es la expectativa de que ellos facilitarán estos procesos. Incluso con sistemas del tamaño de los que se pueden abordar en un contexto académico, se puede lograr que los estudiantes descubran el desafío, aprendan los principios de diseño para el cambio, y experimenten directamente los beneficios.

Clasificación y refinamientos sucesivos. La programación estructurada y la orientada a objetos han mostrado que una forma de atacar la complejidad es organizar aspectos confusos en aspectos incomprendidos de menor magnitud, y repetir el proceso hasta obtener problemas suficientemente pequeños o sencillos que se puedan solucionar directamente.

Tipificación. La idea de atribuirle un tipo (categorización de naturaleza lógico-matemática) a todo, ayuda a producir elementos de software correcto, documentarlos y hacerlos utilizables efectivamente, es otra realización que puede afectar profundamente los cimientos profesionales de la informática. La problemaática y las técnicas de tipificación se repiten a lo largo de todo el campo de la informática, desde la especificación hasta la implementación y la documentación. Nuestra profesión puede alardear de haber hecho la construcción y estudio de los sistemas de tipos, orientados a objetos o no, más allá de lo que lo ha hecho cualquier otra profesión; dominar su poder es una condición necesaria para llegar a ser un buen profesional de la informática.

Diseño basado el concepto de contrato. Estrategia clave para controlar la corrección de un sistema informático. Los módulos interactúan entre sí conforme a especificaciones sobre las características que se puede suponer cumplen los datos, las propiedades que deben cumplir los resultados y la relación entre éstos y los anteriores. La práctica de dotar a los algoritmos, las estructuras de datos, los módulos, y los sistemas, de restricciones precisas, garantías, y propiedades invariantes nos permite un mejor

control de lo que queremos hacer. Una vez dominada, esta habilidad perdurará de por vida.

Manejo de excepciones. Cuando se construyen sistemas informáticos normalmente se piensa en los casos deseables únicamente, pero un profesional debe estar constantemente preocupado de las situaciones anormales también. Únicamente mediante técnicas conceptuales sistemáticas se puede evitar ahogar la parte supuestamente interesante del razonamiento realizado – y también de los programas – en un número inmensamente grande de prevenciones contra los casos excepcionales.

Errores y corrección. Aunque poco se enfatiza este aspecto, buena parte del tiempo diario de los desarrolladores de software se gasta en tratar con cosas que no trabajan como deberían, y que las construyeron ellos mismos u otras personas. Los profesionales de software son expertos mundiales en enmarañarlo todo. Se Debe enseñar al estudiante que éste es un hecho de la vida y mostrarle como lidiar con esta amenaza.

3. Técnicas

En el nivel más mundano, enseñar ingeniería informática también requiere hacer que los estudiantes se familiaricen con técnicas prácticas que han sido probadas como productivas y que son una parte clave del oficio. Algunos ejemplos:

Manejo de configuraciones. Como ya se ha mencionado, los sistemas informáticos debido a su complejidad evolucionan permanentemente, es decir, aún trabajando con especificaciones fijas, surgen diferentes versiones de los diferentes módulos y componentes, que van variando en el tiempo. A una colección coherente de versiones se la llama una *configuración*. En un gran proyecto informático de consideración que involucre varios profesionales, la administración de las diferentes configuraciones constituye un problema no trivial pues desatendido puede generar efectos de “torre de Babel”. A pesar de que ésta es una de las más importantes prácticas que todo proyecto debería aplicar, el manejo de la configuración no se usa tan amplia y sistemáticamente como se debería. El manejo de la configuración está basado en principios simples y está apoyado por las herramientas disponibles.

Administración del proyecto. Esta es un área en la cual tienen deficiencias graves algunos ingenieros informáticos. A pesar de que todavía no se ha encontrado la manera perfecta de hacer esta labor, sí hay aspectos muy importantes relativos a la administración que deberían ser enseñados a todos los estudiantes porque la mayoría de ellos en algún momento ejercerán el rol de administradores de proyectos.

Métricas. A diferencia de las otras ingenierías más antiguas y maduras, que cuentan con normas y métricas precisas para ponderar

materiales, personal, costos, tiempo en el desarrollo de un proyecto, su uso y registro en la informática apenas comienza a sugerirse. En realidad, es una de las técnicas menos usadas en el desarrollo de sistemas informáticos. La mayoría de la literatura actual sobre métricas no es apropiada debido a la falta de teorías formales acerca de qué es lo que se mide y de por qué es relevante hacerlo. Por lo mismo, es importante enseñar a los estudiantes cómo usar las métricas para cuantificar la pertinencia de los proyectos y los atributos de los productos, a evaluar los resultados de los métodos y herramientas mediante criterios objetivos, y a usar herramientas cuantitativas como una ayuda a la predicción y la valoración.

Ergonomía e interfaces con el usuario. Los usuarios de los sistemas informáticos esperan interfaces (instrumentos de interacción) de alta calidad; al igual que para el resto del sistema, la interfaz con el usuario debe ser diseñada apropiadamente, una habilidad que puede ser aprendida.

Documentación. Debido a la azarosa y monumental tarea que constituye el desarrollo de sistemas informáticos por los fenómenos ya explicados relativos a la complejidad computacional, los profesionales de la informática no pueden darse el lujo de limitarse a la construcción de estos sistemas —deben también documentarlo. Un curso sobre escritura de documentos técnicos debería ser parte de cualquier currículo de ingeniería informática.

Interacción con el usuario. La mejor tecnología es inservible a menos que llene las necesidades de los usuarios para los cuales está dirigida. Un buen profesional de la informática debe saber escuchar a los clientes y a los usuarios.

Análisis de alto nivel de un sistema. Para resolver un problema mediante el uso de la informática, se debe primero entender y describir el problema. Esta tarea de análisis es parte integral de la ingeniería informática, y es tan difícil como cualquier otra parte de ingeniería.

Corrección de errores. Los errores y las imperfecciones son una parte integral del trabajo diario del profesional de la informática. Se hacen indispensables técnicas sistemáticas y efectivas de corrección de fallas y errores para cubrir este aspecto del trabajo.

La tarea es tan compleja que no es difícil encontrar otros ejemplos de técnicas poderosas que todo profesional debería conocer y practicar.

4. Aplicaciones

Existen áreas específicas de la Informática como son las relacionadas con los algoritmos y estructuras de datos, los compiladores, los sistemas operativos, las bases de datos, las técnicas de inteligencia artificial y la computación numérica, para mencionar solamente algunas. Estas áreas, a pesar de haber desarrollado tradiciones, técnicas y resultados individuales, no dejan de ser temas de la

ingeniería informática, y deberían ser enseñados de una forma compatible con la visión particular que haya adoptado la institución para encarar la enseñanza de la informática. Es decir, el momento histórico no es propicio para escindir prematuramente la ciencia y disciplina informática en subáreas especializadas. La ventaja es mutua: los temas especializados se benefician de la mejor preparación metodológica de los estudiantes —por ejemplo, los proyectos de programación se pueden concentrar en los aspectos propios del área, en vez de distorsionarse en temas de programación pura porque los estudiantes ya han aprendido habilidades generales de diseño y programación —y éstos a su vez ayudan a cumplir los objetivos de la ingeniería informática proporcionando una abundancia de nuevos ejemplos y aplicaciones.

5. Herramientas

Por el fenómeno tecnológico y cultural sin precedentes que ha ocasionado el desarrollo de la informática, las presiones comerciales para su difusión y vulgarización han dado lugar al fenómeno que en este documento llamamos “enfoque mágico”. Este “enfoque educativo y cultural” pretende presentar la informática al menos para la “población consumidora”, como una técnica que requiere de simples habilidades operativas.

Es preciso cuidar de que las “herramientas” (instrumentos tecnológicos) informáticas de moda en el momento, no determinen la pedagogía. De hecho, científicos destacados de la informática tal como David Parnas tiene palabras bastante fuertes contra la enseñanza de lenguajes específicos y herramientas. Pero si estos aspectos no deben ser la parte central, tampoco se debe ignorarlas o eliminarlas. Es preciso exponer a los estudiantes a algunas herramientas modernas que se usan en la industria. Por ejemplo, los *sistemas operativos* más comunes, los *lenguajes de programación* más comunes y también los mejor diseñados junto con sus ambientes de programación, las *bases de datos* que permitan aprender los diferentes enfoques propuestos, los ambientes *de trabajo colaborativo*, los sistemas de enseñanza virtual (a distancia), y por supuesto además de las aplicaciones de oficina, el uso de internet y del correo electrónico. Esta exposición debería llevarse a cabo con un espíritu crítico, alentando a los estudiantes para que vean los beneficios y limitaciones de estas herramientas – y a pensar en mejores soluciones.

La lista de herramientas no puede y no debería ser exhaustiva. Es mejor seleccionar un puñado de lenguajes de programación y unos pocos productos populares y ayudarle a los estudiantes para que los entiendan a fondo. Si ellos necesitan otras herramientas, podrán aprenderlas por su cuenta a medida que trabajan con ellas. Pero deben ver unas pocas durante sus estudios para que tengan una idea

general de lo que está disponible y qué es lo que los futuros empleadores esperan.

Inevitablemente, algunas habilidades llegan a ser obsoletas cuando el estudiante se gradúa, pero este no es un mal riesgo dado que el estudio de herramientas específicas es sólo un componente del currículo, no su principal objetivo, y es entendido como el estudio de unos pocos ejemplos a la luz de principios más generales. Cuando llegue la ocasión de aprender una herramienta más moderna, el hecho de haber dominado alguno de sus predecesores y entendido sus limitaciones es frecuentemente muy útil.

Estas observaciones aplican en particular a los lenguajes de programación. Si el lenguaje principal adoptado no es uno de los lenguajes dominantes en la industria, el currículo debería incluir unos pocos cursos de servicio (cursos libres) sobre estos estándares de la industria, para que los estudiantes tengan su experiencia en éstos y exponerlos a la variedad de aproximaciones prácticas más populares en la industria. Un buen ingeniero informático debe conocer varios lenguajes, de tal manera que no haya contradicción entre usar un lenguaje para la enseñanza y, permitir que los estudiantes descubran tantos lenguajes nuevos como el tiempo lo permita.

Un programa de educación adecuado debería resistir los intentos del medio por imponer herramientas específicas, en particular lenguajes de programación, sobre la base de una valoración de lo que está en boga en los anuncios de empleo del momento. En esta situación los profesores son responsables de la escogencia de herramientas apropiadas, de acuerdo con su experiencia profesional, y teniendo en cuenta los intereses de los estudiantes, no solamente a corto plazo sino durante el trascurso de su profesión. Por experiencia es sabido que en materia de herramientas y tecnologías lo que hoy está de moda, mañana puede estar olvidado.

VI. EL PROCESO DE ENSEÑANZA-APRENDIZAJE DE LA INFORMATICA

Viviendo, como lo hacemos, en una época en que las aplicaciones de la informática se nos manifiestan de manera omnipresente en prácticamente todas las actividades cotidianas, y en que la revolución de los computadores se da claramente ante nuestros ojos, no deja de ser sorprendente (como se sugirió ya en el numeral sobre herramientas) que la manera de *enseñar* a diseñar y construir dichas aplicaciones —que tan útiles y fascinantes nos parecen— no haya evolucionado consecuentemente, sino que siga realizándose de manera tan desestructurada, autoritaria y tediosa como se tuvo que enfrentar en los primeros días de la informática.

En el presente, se continúa enseñando bajo una concepción demasiado dependiente de los detalles técnicos de las máquinas ('bits' y 'bytes') y de lenguajes con rígidas reglas de gramática y puntuación (la implacabilidad de los puntos y comas). Es decir, no se

ha trascendido aquel enfoque que para el estudiante aparece centrado en los principios de una máquina en particular o en un lenguaje de programación también particular (o en ambos); parecería que no se pueden explicar las cosas sin primero describir, en detalle, una máquina o un medio para comunicarse con ella. En contraste, la revolución informática nunca habría tenido lugar si no se hubieran trascendido los remolinos tecnológicos y lingüísticos de lo específico, si no se hubiera dado rienda suelta un la imaginación y un la creatividad, si no se hubiera apelado un la abstracción, un la generalidad y un la metáfora.

El cálculo lógico: alternativa a la enseñanza tradicional de las matemáticas

De hecho, tradicionalmente, en los primeros cursos de matemáticas para las carreras de ingeniería, matemáticas y ciencias básicas en las universidades (cursos elementales de álgebra, geometría y cálculo) así como en los cursos del bachillerato, se evita el uso de formalismos, tanto en la definición de los conceptos como en las pruebas o demostraciones. Ante el vertiginoso desarrollo tecnólogico es en la escuela donde deben aprenderse las bases del cálculo lógico; resulta más estratégico aprender en la primaria a realizar los primeros cálculos lógicos que aprender a realizar exclusivamente cálculos aritméticos cuando cada vez contamos con máquinas más y más sofisticadas para ayudarnos en las tareas cotidianas.

Posteriormente, (en el caso de la carrera de Infórmatica o Ingeniería de Sistemas), cuando los estudiantes ya manejan definiciones y demostraciones escritas principalmente en lenguaje natural, es posible que se les muestre como formalizar raciocinios informales (cursos de Lógica y Matemáticas Discretas), pero la impresión que les queda es que tal formalización no justifica el esfuerzo. No se les enseña tampoco, a diseñar demostraciones matemáticas. Los estudiantes ven hacer demostraciones y luego se les pide que desarrollen algunas ellos mismos, pero se discuten muy poco o nada los principios o estrategias para diseñar pruebas.

Los resultados de estos cursos no son en general satisfactorios. Aún después de haber pasado por varios cursos de matemáticas, la habilidad para razonar de los estudiantes de estos cursos es en general, muy pobre. Muchos estudiantes continúan temiendo a las matemáticas y a su notación, y el desarrollo de pruebas sigue siendo un misterio para la gran mayoría. En síntesis, los estudiantes de esta carrera no son equipados con las herramientas necesarias para emplear las matemáticas en la solución de problemas nuevos.

Este problema ya ha sido reconocido y enfrentado en otras universidades por figuras muy destacadas de la ciencia de computación y la academia [Lam93], [Gri91] y [Gri95]. Un nuevo enfoque de enseñanza de las matemáticas en particular, para estudiantes de ingeniería, matemáticas y ciencia de computación, ha sido ya propuesto y puesto en marcha con éxito en varias de las

facultades norteamericanas y europeas prestigiosas de ciencia de computacion. La idea de este enfoque es cambiar la forma y énfasis con que se enseña la lógica por primera vez a los estudiantes, en vez de enseñarla mirándola esencialmente como un objeto de estudio en sí mismo, se la considera más bien, una herramienta básica y por tanto, se vuelve muy importante enseñar a usarla con efectividad.

Este nuevo enfoque de enseñanza se basa en la tesis de que las matemáticas y el pensamiento riguroso se pueden enseñar con más efectividad, enseñando primero a diseñar pruebas rigurosas, apoyandose en una lógica formal [Gri94] [Gri98].

Por consiguiente, la selección adecuada del sistema lógico y su método de demostración resulta crítica para el éxito en la aplicación de este enfoque. Una lógica ecuacional, que se base en la igualdad y en la "sustitución de iguales por iguales" de Leibniz resulta ser la más apropiada por poseer las siguientes características.

La lógica ecuacional es fácil de enseñar, porque sigue el estilo del álgebra del bachillerato.

- La lógica ecuacional constituye una alternativa al razonamiento en lenguaje natural. Muy raramente las pruebas en
- la lógica ecuacional remedan argumentos informales en castellano. Por el contrario las demostraciones son calculativas en el sentido de que se calcula, usando las reglas de la lógica, de forma muy similar a como se calcula para resolver un problema de álgebra del bachillerato. Más aún, se pueden poner en práctica principios y estrategias útiles para ayudar a descubrir teoremas y demostraciones.
- El uso riguroso de la lógica ecuacional no conduce necesariamente a complejidades abrumadoras (como es el caso de algunos sistemas lógicos.) Por el contrario, su tendencia es hacia la simplificación. En general, las pruebas calculativas son más cortas, simples y fáciles de recordar que las pruebas informales expresadas en lenguaje natural.
- La lógica ecuacional es muy versatil –se puede extender a una amplia variedad de dominios matemáticos.

La novedad de la propuesta consiste primeramente, en enseñar a realizar demostraciones matemáticas formales mediante un sistema lógico de amplio espectro (incluye de manera natural conceptos fundamentales de la Informática) que da lugar a un método de realizar cálculos y demostrar teoremas muy efectivo y elegante. Posteriormente, y aplicando este sistema lógico como instrumento de demostración y cálculo, estudiar los temas usuales (teoría de conjuntos, relaciones, aritmética, conteo) de los cursos de matemáticas discretas, en coherencia con la forma como se aplican en la informática.

Este método basado en la noción de igualdad y en la sustitución textual de expresiones matemáticas (lógica ecuacional) es análogo al estilo calculativo usado en la enseñanza de los cursos de álgebra elemental en el bachillerato, y por tanto de fácil aprendizaje. A diferencia de la manera tradicional en que se han enseñado estos temas y las matemáticas en general, se centra en enseñar a diseñar demostraciones proporcionando formatos bien definidos para presentar la pruebas y principios o estrategias para diseñarlas. La lógica ecuacional constituye una alternativa al razonamiento en lenguaje natural y no conduce necesariamente a complejidades abrumadoras (como es el caso de algunos sistemas lógicos). Por el contrario, su tendencia es hacia la simplificación. En general, las pruebas calculacionales son más cortas, simples y fáciles de recordar que las pruebas informales expresadas en lenguaje natural.

1. Los pilares de la enseñanza de un arte. El caso del arte de programar un computador

Concebimos la actividad orientada a la construcción de software como la práctica de un arte (desafortunadamente, no es todavía una práctica ingenieril madura, y mucho menos una actividad científica) que se fundamenta en el conocimiento de las ciencias de la computación. Para ser verdaderos artistas en el desarrollo de software, debemos basarnos en tres aspectos principales: la teoría, la técnica y la tecnología. Dado que consideramos la tarea de construir software como un arte, para explicar estos tres conceptos acudiremos a una analogía: la pintura, un arte antiguo y consagrado, que se nos ocurre muy pertinente e ilustrativo.

* La Tecnología

En primer lugar, para desempeñarnos en este arte es necesario un adecuado conocimiento y destreza en el uso de los útiles con que se pinta (pinceles, espátulas, óleos, lienzo, etc.). Similarmente, la construcción de software se realiza por medio del uso de lenguajes de programación, especificación y muchas otras *herramientas*. Estas habilidades constituyen lo que llamamos el aspecto tecnológico del arte. Debido a la novedad e incipiente desarrollo del arte de la computación, la enseñanza de la programación de computadores se centró inicialmente en este primer aspecto; como ya se mencionó, actualmente la adecuada asimilación de la tecnología (continuamente cambiante) sigue constituyendo la directriz principal de esta actividad.

* La Teoría

En segundo lugar, la mera habilidad en el uso de los útiles necesarios para pintar no es suficiente; se hace indispensable la asimilación de conceptos y teorías (la perspectiva, la naturaleza y propiedades de la luz, la sombra, los colores, etc.) sin los cuales no se concibe la pintura como arte. El desarrollo de la ciencia de computación desde los años setenta tuvo un impacto importante en la enseñanza de la programación, con la introducción de importantes conceptos como la noción de programación estructurada, la especificación en términos de pre y poscondiciones, los métodos algorítmicos y la noción de tipo abstracto de datos. Sin embargo, la enseñanza de estos conceptos se realiza todavía, en medio de una multitud de detalles propios de la herramienta tecnológica (el lenguaje de programación) que no facilitan la adecuada comprensión de dichos conceptos, ni su diferenciación de los detalles tecnológicos en cuanto a su importancia y posible abstracción y reutilización.

* La Técnica

Finalmente, el diestro manejo de los útiles y herramientas y el conocimiento de los conceptos y teorías en los cuales se basa la práctica de un arte no son suficientes para formar al artista. Es

indispensable además el aprendizaje y desarrollo de una técnica para el desempeño profesional del arte, a través de una *disciplina* que, en general, se adquiere al lado de un maestro. Este aspecto de la actividad de construcción de software se traduce en la aplicación consciente y disciplinada de principios, conceptos prácticos y métodos tales como un estilo unificado para la construcción de componentes genéricos y reutilizables [Mey88], así como prácticas relacionadas con la adopción de normas y prácticas relacionadas con los principios de calidad tanto a nivel personal como grupal tales como la revisión personal o grupal del trabajo, la medición y la predicción del tiempo y del esfuerzo para realizar un proyecto y el control y manejo explícitos de los errores.

2. La estrategia del “currículo invertido”

La verdad es que el enfoque tradicional con que se enseña la programación de computadores (el método de imitación: "mire y aprenda"; el profesor trabajando desde el ambiente seguro y poco exigente de las generalidades y del pizarrón borrable; el estudiante enfrentándose al concreto y duro mundo del código de programación, donde los errores pululan y no hay muchas guías para prevenirlos, mucho menos para corregirlos; una evaluación implacable basada únicamente en resultados finales), decididamente resulta poco atractivo —por no decir frustrante— para la juventud de hoy, lo cual es grave si consideramos que las generaciones actuales resultan muy tempranamente inducidas a, y fascinadas por, el mundo de las aplicaciones modernas de los computadores.

Hasta hace relativamente poco tiempo existían razones naturales para que la práctica y la enseñanza de la programación, y más globalmente de la construcción de software, fueran muy precarias: limitaciones tecnológicas y ausencia de una conceptualización teórica adecuada y de unos principios de técnica ingenieril bien fundamentados. Sin embargo, ahora que la ingeniería de software está llegando a una madurez largamente esperada, ahora que se cuenta con un desarrollo teórico-metodológico y tecnológico avanzado; ahora, pues, es el momento preciso para dar un vuelco radical en la manera como se enseñan las artes relacionadas con esta profesión.

No hay ninguna razón para posponer la introducción de estos logros en la enseñanza de la programación y otros asuntos relacionados con el software. En lugar de continuar con el enfoque tradicional de presentar primero los comandos y estructuras propias de un lenguaje de programación postergando —para su detrimento— la exposición de los conceptos y métodos fundamentales de la ciencia de computación, proponemos invertir esta tendencia tan fuertemente arraigada, sugiriendo y desarrollando un enfoque educativo de alcances y concepciones más amplias, mediante la elaboración y el diseño de cursos y materiales didácticos apropiados.

Es decir, se trata de concebir todo el proceso como uno de enseñanza-aprendizaje en el cual el estudiante tiene la oportunidad de socializarse activamente en la comprensión global del lenguaje, los conceptos y los criterios primeros para pasar luego a los detalles técnicos propios de la implantación de los mismos en un lenguaje de programación. Así, los fundamentos teóricos se reflejan directamente en los aspectos técnicos y éstos en forma continua, se hacen más claros por la comprensión de los conceptos fundamentales.

Esta estrategia es preferible a una más tradicional y conservadora en la cual se enseña primero un método obsoleto de programación para luego desaprenderlo y finalmente introducir concepciones modernas de la programación como por ejemplo, la orientación a objetos. Tal como lo señala Bertrand Meyer,

"Los profesores a veces tienen una tendencia inconsciente a aplicar una idea que solía ser popular en biología: que la ontogenia (el desarrollo del individuo) repite la filogenia (el desarrollo de las especies) -un embrión humano, en las varias etapas de su desarrollo vagamente se asemeja a una rana, a un cerdo etc. Traducido a la educación, esto significa que un profesor que primero aprendió Algol, continuó con diseño estructurado y finalmente descubrió la orientación a objetos, quiere llevar a sus estudiantes por el mismo camino. Existe poca justificación para tal enfoque que, traducido a la educación primaria, significaría que los estudiantes primero aprendieran a contar con la numeración romana, sólo para que posteriormente les presentaran "metodologías" más avanzadas, tales como los números arábigos. *Si usted cree que conoce el enfoque correcto enséñelo de una vez*." [Mey94a]

Es más, la ya mencionada madurez de la tecnología y de los métodos de construcción de software nos proporcionan una oportunidad formidable para enriquecer la enseñanza de esta disciplina con dimensiones educativas no contempladas previamente, al menos de manera consciente. Nos referimos a una educación en informática orientada por la apreciación estética, el incentivo de la creatividad y del trabajo grupal, armonioso y productivo.

En cuanto a la apreciación estética y a la actividad creativa se refiere, podemos seguir el ejemplo de algunos excelentes maestros y divulgadores entre nuestros colegas matemáticos, siendo conscientes de la importancia del placer estético y la satisfacción espiritual asociados al gozo impulsor de la creatividad. Este profundo gozo ha sido considerado por algunos educadores como el objetivo principal de la educación -más aún: el fin primordial de la vida humana [Hun70]. En *La Educación del Hombre Integral* L.P. Jacks escribe:

¿Cuál es entonces la vocación del hombre integral? Hasta donde se me ocurre, su vocación es la de ser un creador: y si usted me pregunta: ¿Creador de qué?, yo respondo: creador de valores reales. ...Y si usted me pregunta: ¿a qué motivo se puede apelar, en qué fuerza directiva se puede confiar, para sacar a la luz el elemento creativo en el hombre y en la mujer?, sólo hay una respuesta que puedo dar, pero la doy sin ninguna reserva ni titubeo: el amor por la belleza, innato en todos nosotros, pero reprimido, sofocado, frustrado en la mayoría de nosotros. . . . [Jac]

Se trataría entonces, de integrar en la actividad de construir software, la dimensión estética involucrada en su creación y los valores del hombre integral que en este artículo consideramos son aquellos de alguien que trabaja en comunidad. Además, no hay que olvidar que en la actualidad, dados el tamaño y la complejidad de las tareas a realizar, el desarrollo individual de sistemas informáticos resulta impracticable, lo cual nos obliga también a plantear la necesidad de mudar el énfasis en el trabajo individual por una cultura de creación colectiva.

Orientar la educación para la construcción de componentes informáticos (software) en comunidad, estamos convencidos de que no sólo es posible, sino también necesario, aunque ello implique abordar ciertas cualidades humanas que debe poseer quien quiera seriamente colaborar en un grupo de desarrollo. Constituye un desafío comprometerse con aspectos humanos que se requiere sean compartidos por profesionales, profesores y estudiantes; sin embargo, es un desafío posible y, como hemos dicho, necesario. Tradicionalmente, no nos sorprendían los ingenieros solitarios desarrollando software en un garaje y nunca habíamos considerado que el trabajo en grupo podría convertirse en algo central a nuestra actividad; sin embargo, si hoy queremos avanzar en esta disciplina, debemos asumir concienzudamente el reto de prepararnos y preparar a los demás para lograrlo, siendo creativos y productivos, no sólo individualmente sino buscando también la posibilidad de interactuar de manera constructiva y armoniosa.

Dentro de la gran cantidad de cualidades humanas que podrían enumerarse para la construcción colectiva de software de una forma efectiva y eficiente, mencionamos algunas que surgen inmediatamente: capacidad para reconocer y aceptar los propios errores, tolerancia hacia los errores de los demás, sentido de identidad y pertenencia a un grupo, y por lo mismo, sentimientos de orgullo, honor, lealtad, colaboración, solidaridad; además de cualidades personales, tales como disciplina, autonomía, concentración, paciencia (tolerancia al fracaso), determinación y alegría de vivir.

Hablamos de actividades comunitarias de construcción colectiva de sistemas informáticos en las cuales, la búsqueda y señalamiento de los errores cometidos por el compañero o colega, que se asimilan como parte de un trabajo productivo y de colaboracion con la persona que comete el error (como sucede en equipos profesionales de construccion de software), serían altamente formativas y cambiarían muchas actitudes perjudiciales para el propio aprendizaje y para el desempeño profesional comunitario. Para ampliar lo aquí presentado, consideramos que las ideas propuestas por W. S. Humphrey en su *Personal Software Process* [Hum96] son altamente relevantes.

Una de las grandes promesas de la ingeniería de software moderno es la reutilización. El llamado currículo invertido para la enseñanza de

la Informática, basado en la “progresiva apertura de cajas negras”, consiste en que los estudiantes primero usan herramientas poderosas y componentes informáticos dentro de sus propias aplicaciones, y entonces progresivamente las van desarmando para ver cómo están hechas las cosas, modificarlas, y extenderlas. El proceso educativo parte desde la visión del usuario de tecnología aproximándose paulatinamente, a medida que a la vez, se adentra en el develamiento y comprensión de los “misterios” (fundamentos teóricos) a la visión del constructor, pero enfocándose desde el principio en ejemplos interesantes y posiblemente grandes.

Esto tiene varios beneficios. Desde el comienzo de su carrera los estudiantes tienen contacto con programas interesantes y emocionantes, visualizan los resultados de sus programas de forma amena e inmediata. Esto cautiva a estudiantes que desde pequeños han usado juegos electrónicos y computadores personales, y a quienes ya no cautivan ni impresionan los tradicionales ejemplos de un curso de introducción a la programación. Tratar de que los estudiantes estén emocionados es pedagógico, no demagógico.

Además, los estudiantes aprenden sobre la necesidad y beneficios de prácticar y aplicar principios y estrategias tales como la abstracción y el ocultamiento de información.

Todos estos conceptos se pueden explicar mediante discursos, pero una estrategia pedagógica que realmente muestra la necesidad de ciertas prácticas de software, consiste en que el estudiante trate de modificar un módulo o componente informático y encuentre que carece de una especificación apropiada, de contratos formales, y descripciones correctas de lo que debería suceder en casos excepcionales.

3. Enseñar haciendo

Los científicos y profesionales de la informática no son en general conocedores y menos expertos o investigadores de las diversas corrientes pedagógicas, sinembargo, parece existir un consenso tácito dentro de esta comunidad, sobre la relevancia del aprendizaje por competencias. Se considera estratégico, que además de los cursos formales, cualquier currículo en informática incluya el desarrollo de proyectos. Frecuentemente, hay insuficiente preparación para afrontar los verdaderos retos del desarrollo profesional en informática, que normalmente incluye sistemas de gran envergadura, grupos grandes de desarrolladores, modificación de sistemas construídos por otros, e interacción con usuarios finales.

Es imposible, y tal vez no deseable, que una universidad trate de simular completamente todas estas circunstancias. Después de todo, una universidad no es una compañía, y no debería tratar de serlo. Pero se debe preparar al estudiante para los retos reales que debe enfrentar. El proyecto académico estándar no es suficiente para esto. Una técnica importante es el proyecto de “larga duración”, que los

estudiantes desarrollan durante un periodo de tiempo superior al de un semestre, típicamente, a lo largo de un año completo. Este debería ser un proyecto de grupo que incluya aspectos de análisis, diseño, implementación y prueba. También debería incluir la reutilización, comprensión, modificación, y extensión de sistemas y componentes ya existentes. La mejor forma de alcanzar este último objetivo es emprender un proyecto que se efectúe varios años seguidos, haciendo que cada nuevo grupo de estudiantes tome los resultados del grupo precedente, y los mejore y los extienda. Ya en algunas universidades se han dado experiencias exitosas en este sentido, a pesar de las dificultades que tiene acoplar un esquema de trabajo como este, con el estilo tradicional de trabajos semestrales y asignación de notas de forma generalmente individual.

Los currículos de Informática deben mantener un balance entre lo operacional y lo conceptual, entre los principios y las técnicas. No deben sacrificar un aspecto en favor del otro. El reto de la academia es crear un programa de enseñanza e investigación que sea al mismo tiempo, serio, ambicioso, atractivo para los estudiantes, técnicamente actualizado, firmemente arraigado en la práctica y científicamente interesante.

VII. HACIA UNA NUEVA APROXIMACIÓN AL CURRICULO EN INFORMÁTICA

A lo largo de este documento se han mencionado aspectos fundamentales que se deben tener en cuenta para proponer una nueva visión de la educación en tecnología informática; y que deben conducir a revisar la manera en que tradicionalmente se abordan los currículos. Se han tratado ideas novedosas como:

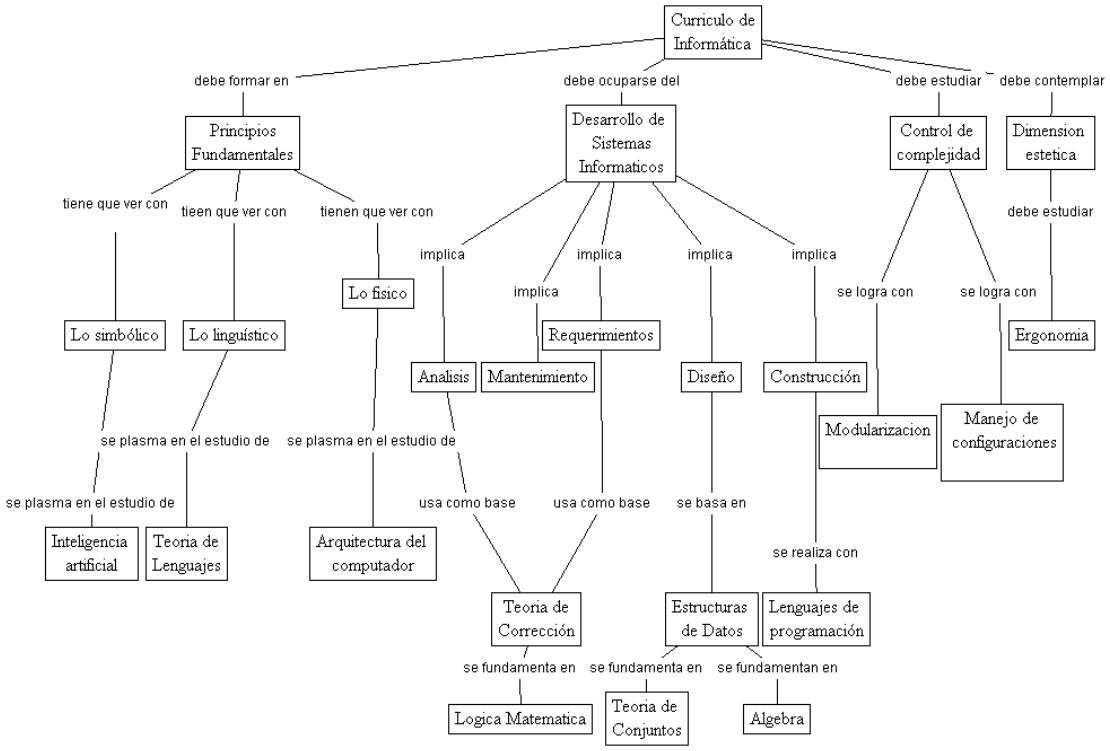
- Comprensión de principios fundamentales de la disciplina.
- Construcción de Software en Comunidad.
- dimensión estética y creativa a la construcción de sistemas informáticos.
- Manejo de la complejidad.
- Herramientas, técnicas y prácticas indispensables en la formación en Informática.

El siguiente mapa conceptual, desarrollado con base en todos los aspectos anteriormente mencionados, integra las diferentes dimensiones de la informática e ilustra los contenidos de formación que podrían ser base para un currículo en informática como son :

- Teoría de Lenguajes
- Inteligencia Artificial
- Teoría de Corrección

    - Lógica Matemática
- Arquitectura del Computador
- Estructuras de Datos
    - Algebra
    - Teoría de Conjuntos

Y otros.



Sin embargo, a nivel pedagógico , consideramos más importante que delucidar los contenidos precisos de formación (el qué?), el proponer una metodología (el cómo?) para generar las competencias necesarias y que el estudiante logre no solo apropiarse de estos contenidos sino tener la capacidad de asimilar nuevos contenidos a medida que la disciplina informática evoluciona. Estas competencias que subyacen a todo el mapa conceptual y que constituyen una herramienta esencial para la llamada educación continua, se pueden resumir en las siguientes: (ver [Den92])

- Escucha: Desarrollar la capacidad de identificar los requerimientos de su entorno y formar alianzas con miembros de la comunidad para satisfacer estos requerimietnos.
- Cabalidad: Rigor en el logro y cumplimiento de objetivos y compromisos, puntualidad en su entrega, y satisfacción por parte de la comunidad en cuanto a estos logros y compromisos.
- Aprendizaje: Actitud positiva hacia el aprendizaje y hacia el mejoramiento continuo en la disciplina.

Cada vez se hace más importante debido a la rapidez en el cambio de contenidos de una disciplina (se estima que el conocimiento en el mundo comenzará a doblarse en tan solo 5 años muy pronto) el basar la formación menos en contenidos y más en competencias.

VIII. PROPUESTA METODOLÓGICA Y DE EVALUACIÓN

Los planes de estudio tradicionales de la educación superior colombiana, basados en materias semestrales y en esquemas tradicionales de enseñanza han probado no ser adecuados para lograr las nuevas habilidades arriba mencionadas. Sinembargo, como ya se mencionó existe una corriente entre los pedagogos hispanoamericanos principalmente dirigida a la educación primaria y media denominada aprendizaje por competencias que parece coincidir en varios de los presupuestos que comparte la comunidad informática internacional sobre la enseñanza de la informática y educación de futuros profesionales en esta disciplina (ver [Bog00]).

Dentro del marco estas concepciones [Den92], [D&M02], se propone basar el currículo alrededor de lo que podrían llamarse "exhibiciones de proyectos", en una exhibición un estudiante debe demostrar públicamente habilidades para pensar, actuar de manera rápida, integrar conocimientos de su disciplina, de las matemáticas, la ciencia, las artes, la historia y la filosofía, etc.

- El estudiante tendría desde el primer semestre un tutor personal y tendría que responder a tres exhibiciones a lo largo de su carrera.
- Las exhibiciones (proyectos) tendrán una duración mínima de un año y máxima de dos años , la facultad sería responsable en este caso de brindar al estudiante un ambiente que le permita lograr un conocimiento profundo del tema de la exhibición, dejando la libertad al estudiante de construir su camino, organizar su trabajo y tomar sus propias decisiones.
- La facultad debe tener nexos con el medio externo que permitan que los estudiantes abran sus ojos a las comunidades donde viven y participan con miras al desarrollo cabal de su proyecto.
- La facultad debe estar en capacidad de programar cursos y seminarios electivos que el estudiante podrá seleccionar (muy posiblemente se requieran cursos de otras facultades) destinados a que los estudiantes profundicen en temas que son importantes para el desarrollo de su proyecto. Cada uno de estos cursos o seminarios debería tener proyectos y exhibiciones pequeñas que sirvan como entrenamiento a la exhibición final.
- Los cursos y seminarios serán electivos no tendrán ni créditos ni nota pero seran la base para una culminación exitosa de los proyectos.

- La evaluación estaría centrada en las exhibiciones en las cuales el estudiante será evaluado junto con su tutor y ambos serán responsables del éxito o fracaso de esta. Para ello se contará con un "jurado" serio que evalúe la exhibición en sus diferentes aspectos y sobre todo pueda señalar fallas con vistas a la mejora del trabajo.El estudiante y su tutor podrán realizar nuevas exhibiciones hasta lograr un proyecto de alta calidad.

Los proyectos podrían tener las siguientes temáticas:

1. Exhibición 1: Proyecto básico: Dirigido a conocer los fundamentos de la informática.
2. Exhibición 2: Proyecto Intermedio: Dirigido a la solución de un problema del entorno con tecnología informática.
3. Exhibición 3: Proyecto Avanzado: Dirigido a la solución de un problema educativo específico en una institución de educación media con ayuda de Tecnología.

IX. DOMINIO TÉCNICO OPERATIVO

Como ya se ha señalado, existen fuertes tendencias e intereses por centrarse en los aspectos puramente técnicos y operativos de la informática (“enfoque mágico”); no obstante, dada su aplicación desbordante en todos los campos de la actividad y el saber humanos, es importante pronunciarse también sobre la enseñanza de estas destrezas operativas, pues la reflexión pedagógica sobre estos temas es muy escasa e incipiente.

De la misma manera como se señaló la tendencia tradicional a enseñar la programación de computadores centrándose en los aspectos puramente técnicos, la enseñanza actual de habilidades operativas informáticas presenta grandes deficiencias precisamente, aunque parezca contradictorio, en dejar de lado la comprensión de conceptos básicos, o al menos, carecer de modelos analógicos que guíen y proporcionen una comprensión mínima de la naturaleza, propósito, utilidad, posibilidades y limitaciones de los instrumentos o artefactos informáticos a sus futuros operarios.

Posiblemente ésta es la razón por la que se presenta una gran resistencia a cambiar o aprender a usar nuevas y desconocidas herramientas informáticas. Un aprendizaje basado en la memorización de instrucciones con sus respectivas digitaciones y manipulaciones del teclado y el “ratón”, desprovistas de un sentido mínimo de las convenciones y racionalidad en que se soportan, resulta muy tedioso y fácil de olvidar por decir lo menos.

De hecho, prácticamente con todos los artefactos y herramientas tecnológicas, surgen como fruto de inspiración en analogías que se dan entre fenómenos ya existen en la naturaleza u otros artefactos ya conocidos y la función o el artefacto que se quiere suplir o construir.

Del mismo modo, estas analógias que inspiraron la concepción o desarrollo de un útil o artefacto informático deben también guiar la enseñanza de su naturaleza y manipulación. Como ilustración basta mencionar como los sistemas operativos modernos basados en interfases gráficas interactivas (v.gr. McIntosh, Windows) se inspiran en el modelo de la “superficie de un escritorio” (top desk modelo); asi como también, las redes informáticas (internet) se inspiran en el modelo de un libro cuyas *páginas* son aportadas por los diferentes “sitios” de la red. De hecho la terminología técnica de estos productos tecnológicos comparte varios elementos propios de la terminología respectivamente de los implementos de un escritorio de oficina, y de los libros y textos.

X. INTEGRACIÓN CON OTROS DOMINIOS TECNOLÓGICOS

Desde los puentes romanos y los aviones jet hasta los motores a vapor y los equipos de CD, es privilegio de los ingenieros y arquitectos combinar visualizaciones científicas con posibilidades técnicas para crear útiles y elegantes artefactos. Los ingenieros y diseñadores en general, obtienen grandes satisfacciones por la utilidad y belleza de sus *diseños*. El diseñador exitoso está familiarizado con las bases científicas de su campo, la tecnología de los componentes utilizadas, la ecología e impacto social de la problemaática que lo ocupa, así como un olfato para las aplicaciones visionarias. Por ejemplo, para construir un avión, más vale entender la física del movimiento, las propiedades estructurales del aluminio, el tamaño y condiciones vitales de los pasajeros. La física del movimiento requiere a su vez, la maestría de las matemáticas, en particular del cálculo. La informática como disciplina de orden tecnológico comparte con las demás ingenierías y disciplinas de diseño, todas estas concepciones, enfoques y problemaáticas.

Sinembargo, como ya hemos señalado, la informática es también una ciencia relacionada con la lógica y la lingüística. Desde este punto de vista, difiere totalmente de los demás dominios tecnológicos en cuanto a que no soporta sus desarrollos en las ciencias naturales, y en consecuencia, los formalismos matemáticos en que se apoya, no son continuos (las tolerancias en los cálculos, tan comunes en las ingenierías tradicionales, no existen en el dominio informático). En este sentido las matemáticas en que se apoya la informática tiene más que ver con manipulaciones simbólicas (algebra, lógica) y finitas (investigación de operaciones). Por otra parte, en la medida en que se puedan concebir “megatextos dinámicos” que describan o modelen fenómenos de una cierta actividad, disciplina o ciencia, la informática interactúa con todos los campos de la actividad y el saber. Desde este punto de vista, Una de las características que hace diferente a los computadores de los demás productos tecnológicos es su "*programabilidad*".

En un nivel lingüístico (simbólico) elemental. todos los computadores son programados más o menos de la misma manera. Un programa, junto con las cantidades sobre las cuales opera, es una larga serie de unos y ceros, llamados "bits" (dígitos binarios). La operación del computador consiste esencialmente en repetidamente inspeccionar esos bits, y ocasionalmente cambiarlos. Algunos de esos bits estan conectados a otros dispositivos y reflejan lo que está sucediendo dentro del computador, abriendo o cerrando válvulas, encendiendo o apagando luces, o moviendo brazos. El ingeniero de computadores (hardware) estudia métodos de representar los bits mediante dispositivos físicos y de implantar las diversas operaciones. La ausencia de estructura en la serie de bits permite que las operaciones sean realizadas por una máquina razonablemente simple. También hace posible, usar esta máquina para un *amplio rango de aplicaciones* porque difícilmente existen limitaciones predeterminadas. El ingeniero informático estudia métodos de construir estructuras a partir de secuencias desestructuradas de bits para *representar los objetos* que jueguen un papel en la aplicación entre manos, y *diseña algorítmos* para llevar a cabo las operaciones relevantes. En este sentido los profesionales de la informática son los equivalentes modernos de los antiguos escribas cuya labor era interpretar y transcribir lo que la sociedad de su tiempo deseaba comunicar o expresar .

BIBLIOGRAFIA

[Bog00] D. Bogoya M y otros: *Competencias y proyecto pedagógico*. Universidad Nacional de Colombia, Mayo 2002.

[Cho56] Noam Chomsky: "Three Models for the Description of Language," IRE Transactions on Information Theory 2(3):113-124, 1956.

[D&M02] O. C. Diaz, L. F: Marín: “Curriculo Integrado para la educación en Tecnología e Informática, Objetivo: Fundamentación Conceptual”, Universidad Pedagógica Nacional, Agosto 2002.

[Dav65] Davis, M: *The Undecidable*. Raven Press, Hewlett, N.Y, 1965.

[Den92] P. Denning: “Educating a new Engineer”, Communications de the ACM, vol 35 # 12, 1992.

[Fr56] Friend, E.H. "Sorting on Electronic Computer Systems," Journal of the Association for Computing Machinery 3:134-163, 1956.

[Gar79] M.R. Garey, D.S. Johnson: *Computers and Intractability*. W.H. Freeman, San Francisco, Calif, 1979.

[Gri91] Gries, D., Calculation and discrimination: a more effective curriculum, CACM Vol 34, No. 3, Marzo 1991.

[Gri94] Gries, D. Schneider, F.B., A logical approach to discrete math, Springer Verlag, 1994.

[Gri95] Gries, D. Schneider, F.B., Teaching Math more effectively, through calculational proofs, AMM, Octubre 1995.

[Gri98] Gries, D. Schneider, F.B., An introduction to teaching logic as a tool, http://simon.cs.cornell.edu/home/gries/Logic/Introduction.html

[Hoc97] Hochbaum, Dorit S., ed. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, Mass, 1997.

[Hol91] Holzmann, G.J. *The Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, N.J, 1991.

[Huf54] Huffman, D.A. "The Synthesis of Sequential Switching Machines," Journal of the Franklin Institute 257:161-190, 275-303, 1954.

[Hun70] H. E. Huntley: *The Divine Proportion: A study in matemática beauty*, Dover Publications, inc., New York, 1970.

[Hum96] W. S. Humphrey: *Personal Software Process*, Software Engineering Institute Series, Addisson Wesley, 1996.

[Jac] L. P. Jacks: *The Education de the Whole Man*, Universidad de London Press, Portway Reprints.

[Kle36] S. Kleene, "General Recursive Functions of Natural Numbers," Mathematische Annalen 112:727-742, 1936.

[Knu68] D.E. Knuth: The Art of Computer Programming, 4 vols. Addison-Wesley, Reading, Mass, 1968.

[Knu76] D.E. Knuth: "Big Omicron, Big Omega and Big Theta," SIGACT News 8:18-23, 1976.

[Lam93] Lamport, L., How to write a proof, Systems Research Center (SRC), February 14th 1993.

[McC43] McCulloch, Warren S., and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity," Bulletin of Mathematical Biophysics 3:115-133. Reprinted in Warren S, 1943.
[Mea55] Mealy, G.H. "A Method for Synthesizing Sequential Circuits." Bell System Technical Journal 34:1045-1079, 1955.
[Mey01] B. Meyer: “Software Engineering in the Academy”, Software IEEE, May 2001.
[Mey97] B. Meyer: *Object Oriented Software Construction*, Prentice Hall, New York, 1988. Second Edition 1994.
[Mey94] B. Meyer: *Reusable Software: The base object oriented libraries*, Prentice Hall, 1994.
[Moo56] Moore, E. "Gedanken Experiments on Sequential Machines," Automata Studies, Claude E. Shannon and J. McCarthy, eds. Princeton University Press, Princeton, N.J, 1956.
[Neu95] Neumann, P.G. *Computer-Related Risks*. Addison-Wesley, Reading, Mass, 1995.
[Rab59] Rabin, M.O., and D. Scott: "Finite Automata and Their Decision Problems," IBM Journal of Research and Development 3:114-125, 1959.
[Sha48] Shannon, Claude E. *A Mathematical Theory of Communication*, Bell System Technical Journal 27:379-423, 623-656, 1948.
[Tou90] S. Toulmin: *Cosmopolis: The Hidden Agenda de Modernity*, Universidad de Chicago Press, 1994.
[Tur36] Turing, A.M. "On Computable Numbers with an Application to the Entscheidungsproblem," Proceedings of the London Mathematics Society 2:230-265, 1936.
[V&W86] Vardi, M.Y., and P. Wolper. "An Automata-theoretic Approach to Automatic Program Verification," pp. 322-331 in Proceedings of the Symposium on Logic in Computer Science. IEEE Press, New York, 1986.