Programming as Theory building (Peter Natur)

- Busca contribuir al entendimiento de qué es la programación
- Sugiere que debería ser tratada como una actividad mediante la cual el programador desarrolla una teoría, un conocimiento, del tema a tratar
  - En contraste de la perspectiva contemporánea más común de que se trata de simple producción de texto
- Destaca la importancia de entender apropiadamente qué es la programación, para no malinterpretar las dificultades que surgen y determinar apropiadamente cómo lidiar con ellas, evitando así frustraciones innecesarias
- Programación: diseño e implementación de soluciones programadas, en particular la relación/matcheo de una parte significativa de algo del mundo real a la representación formal de un programa en la computadora.
  - Esto implica que cambios en el mundo real deben verse reflejados en modificaciones en el programa
- En este contexto, programación para un programador debe ser la construcción de conocimiento, en posesión inmediata al mismo, con documentación y otros siendo productos secundarios/auxiliares
- Acá tal vez convenga repasar los ejemplos
- **Noción de Ryle de teoría**
  - Debemos caracterizar qué es el conocimiento mencionado antes
  - Se propone tratar la misma como una teoría, así como la define Ryle
  - Análisis que distingue la actividad intelectual difiere, y va más allá de, aquella que es meramente inteligente
  - Aquel que posee una teoría es capaz de hacer ciertas cosas, pero además proveer con explicación, responder dudas, etc.
  - En el comportamiento inteligente, el individuo demuestra, no necesariamente un conocimiento particular de hechos, sino ser capaz de hacer algo de manera correcta de acuerdo a ciertos criterios, y de poder identificar fallas y corregirlas de acuerdo a dicho criterio
  - El comportamiento inteligente no se define por un conjunto de reglas, ya que esto causaría una regresión al infinito
    † Aunque la capacidad de seguir reglas en sí misma es una actividad inteligente
  - Lo que caracteriza a la actividad intelectual, y lo que la eleva más allá de la meramente inteligente, en cambio es esta construcción de una teoría, entendida como el conocimiento necesario para no solo hacer cosas de manera inteligente, pero para también explicarlas y discutir acerca de ellas
  - La construcción de la teoría y el deseo de tenerla son cosas que ocurren a la vez
  - Aquel con la teoría debe ser capaz de relacionar conceptos e identificar similitudes, etc.
- **La teoría a ser construida por el programador**
  - La teoría a ser construida es el cómo entes (esencia?) de la realidad van a ser representados por un programa

- El conocimiento obtenido mediante la obtención de esta teoría trasciende la que puede obtenerse de cualquier manera auxiliar en al menos tres maneras
  - La justificación de cómo el modelo se relaciona a la realidad, el mapeo que se realizó, y la justificación de la decisión de lo que no era esencial y quedó fuera del programa son todas cosas que solo el programador con la teoría es capaz de aportar
  - Puede explicar el porqué el programa en sí es lo que es de manera directa e intuitiva debido a su conocimiento del "mundo", los métodos y patrones utilizados para el desarrollo del mismo son justificados por que el programador decidió que los mismos eran relevantes a la situación
  - El programador es capaz de responder constructivamente a demandas por modificaciones para poder soportar los cambios en la realidad (o la perspectiva de la misma) de una nueva manera

- **Problemas y Costos de Modificación de Programas**
  - Un programa siempre va a terminar siendo modificado
  - Esto está inherentemente relacionado a costos, y se desea buscar la manera de minimizarlo
  - El hecho de que modificaciones a poco costo es siquiera posible es algo cuestionable, ya que en principio esta esperanza se basa en ver a la programación como simple producción de texto, y que el programa no se trata de nada más que el texto
    - En esta visión de theory building todo este argumento es falso, acá no hay soporte para la idea de que modificaciones a bajo costo sean posibles
  - Otro argumento a analizar es el de flexibilidad del programa, donde se agrega en un principio funcionalidad extra a un programa que no es inmediatamente necesaria, pero que probablemente lo terminen siendo
  - En general, ésta flexibilidad viene a un gran costo, ya que su existencia implica que debe ser apropiadamente testeada, diseñada, etc; a pesar de no haber certeza acerca de su utilidad
  - Flexibilidad solo naturalmente alcanzable y no forzada (Implica costo innecesario)
  - Aquel con la teoría es el único capacitado para determinar las capacidades de la solución actual y las nuevas demandas dadas por las modificaciones
  - Esta perspectiva explica el porqué de la decadencia del programa luego de modificaciones hechas por programadores sin la teoría
    - Si se trata a la programación como mera producción de texto, entonces hay múltiples posibles cambios al programa, todos técnicamente correctos. Sin embargo, desde la theory building view, estos programas probablemente se vean altamente diferentes, de acuerdo a qué tan natural fue la alineación/extensión a la teoría original.

- **Vida, Muerte, y resucitación del Programa**
  - **Vida del programa**
    - Un equipo activo que posee la teoría del programa se encuentra en posesión del mismo, y controla todas las modificaciones que se le hacen al mismo
  - **Muerte del programa**
    - Cuando el equipo se disuelve, la muerte se hace evidente cuando demanda por modificaciones no puede ser respondida de manera inteligente (No debería ser intelectual? Ni él es consistente)

- **Resucitación del programa**:
  - † reconstrucción de su teoría por un nuevo equipo de programadores
- Es de esta manera que la extensión de la vida de un programa depende de la adopción de nuevas generaciones de la teoría del mismo
  - † Para lograr esto, es necesario que el nuevo programador tenga la oportunidad de trabajar con el equipo que ya posee la teoría
- Desde la perspectiva de este paper, la resucitación de un programa a partir de la documentación del mismo es imposible, y como mínimo, extremadamente costosa
- Una preferible alternativa sería descartar el programa y empezar desde 0, logrando que el nuevo equipo aprenda una nueva teoría, para bien o para mal

- **Métodos y construcción de la teoría**
  - Método de programación: serie de reglas de trabajo para el programador, dictando cómo hacer diferentes tareas, ordenarlas, etc
  - Esto ya entra en contraste con la noción de teoría, ya que el método proclama que el desarrollo se puede llevar a cabo mediante una serie de pasos ordenados, mientras que la teoría inherentemente no puede ser dividida en partes ni ser ordenada
  - Es por esta observación que se puede concluir que no existe un método "correcto" para el desarrollo de software
  - Esta conclusión parecería contradecir dos opiniones comúnes acerca del desarrollo de software
    - † La primera es que el desarrollo de software debería estar basado en el método científico, y por lo tanto emplear la metodología. Sin embargo, esta supuesta contradicción se basa en suponer que el método en sí es existe y es útil, lo que es debatible
    - † La segunda contradicción es el hecho de que estudios realizados parecerían indicar que específicos métodos sí son superiores a la hora de desarrollar, sin embargo, este argumento se basa en la suposición de que estos estudios fueron realizados de manera correcta, lo que no es cierto.
  - Donde los métodos sí son útiles es en el contexto de la educación del programador
- Toda esta perspectiva lleva a un repensamiento de la importancia del programador, y cómo el mismo no es solo un engranaje más en la cadena de producción

No Silver Bullet (Frederick P. Brooks Jr)

- Todo el desarrollo de software involucra tareas **esenciales** y **accidentales**:
  - Esenciales: la elaboración de las estructuras conceptuales que componen a la entidad abstracta del software
  - Accidentales: La representación de estas entidades en lenguajes de programación y el mapeo de estas en lenguaje de máquina que se adhieran a diferentes restricciones de velocidad y espacio
- La gran mayoría de las ganancias en productividad de software fueron dadas por remover las barreras artificiales que hacían las tareas accidentales demasiado difíciles (restricciones de hardware, lenguajes de programación rancios)
- Llegó la hora de considerar las partes esenciales del desarrollo, sugiriendo
  - Hacer uso del mercado para evitar construir lo que puede ser comprado

- Hacer uso de prototipado rápido iterativamente para establecer requerimientos de software
- Crecer software orgánicamente, agregando funcionalidad a sistemas a medida que son usados y testeados
- Identificar los grandes diseñadores conceptuales de la generación

- Se busca una **"silver bullet"** para proyectos de software, algo que haga que los costos de software bajen tanto como los de hardware
- Sin embargo, dicha mejora no se ha visto, una que ofrezca una mejora de al menos una orden de magnitud en relación al hardware en cuanto productividad, simplicidad, fiabilidad
- Es muy poco probable que puedan ocurrir mejoras al mismo nivel que lo que ocurrió para hardware
  - Lo que pasó para el hardware es primero que nada, una anomalía en sí
- Para considerar qué tasa de progreso se podría llegar a esperar, hay que primero analizar sus dificultades
  - Las **esenciales**, inherentes en la naturaleza del software
  - Las **accidentales**, las no esenciales
- **Consideremos primero las esenciales**:
  - La esencia de una entidad de software está dada por una interconexión de conceptos: conjuntos de datos, relaciones entre objetos, algoritmos, funciones, etc. Altamente precisa y detallada.
  - La parte difícil del desarrollo de software es la especificación, diseño y testeo de este ente conceptual, no el labor de representarlo y testear la fidelidad de la representación.
  - Si esto es cierto, entonces el desarrollo de software siempre va a ser difícil
  - Consideremos las propiedades inherentes de esta esencia irreducible: complejidad, conformidad, posibilidad de cambiar (changeability), invisibilidad

    † **Complejidad**

    ‡ Las entidades de software son bastante complejas respecto de su tamaño, ya que no hay partes que se parezcan (pues si las hubiese, se podría hacer uso de alguna abstracción)

    ‡ Una ampliación de estas entidades es por esto necesariamente un incremento en la cantidad de diferentes elementos y, como generalmente, los elementos interactúan entre sí en una manera no lineal, llevan a un incremento en complejidad mucho más que lineal

    ‡ La complejidad de software es una propiedad esencial, por eso es que abstraer complejidad también lleva a abstraer su esencia

    ‡ La razón por la que los matemáticos y físicos pudieron simplificar sus modelos de la complejidad es porque la misma no era esencial

    ‡ Es por esta complejidad inherente que:

    - Surge la dificultad en comunicación en los equipos, causando fallas en el producto, retraso, etc.
    - Surge la dificultad de enumerar, y entender, todos los posibles estados de un programa, lo que lleva a la falta de confiabilidad
    - La dificultad de entender y llamar funciones, causando que programas sean difíciles de usar

        - La dificultad de extender programas con nueva funcionalidad sin efectos secundarios inesperados
    - **Conformidad**
        - A diferencia de los físicos, que pueden tener fe de que hay principios unificantes de los problemas a mano, el desarrollador de software no tiene ese lujo, ya que gran parte de la complejidad con la que debe lidiar es a mano del capricho humano
    - **Changeability**
        - El software está constantemente siendo expuesto a la necesidad de cambios
        - En parte porque el mismo representa su funcionalidad, en parte porque es infinitamente modificable.
        - Todo software exitoso es modificado, esto se debe a dos procesos
            - Si un software es identificado como útil, los usuarios intentan usarlo en los bordes, o incluso más allá de, el dominio original, los usuarios presionan para que funcionalidad sea extendida
            - El software exitoso debe ser adaptado a la máquina en la que es usado, que va a cambiar respecto de la máquina para la cual fue originalmente diseñado
    - **Invisibilidad**
        - El software es inherentemente no visual, por lo que no puede haber una eficiente representación gráfica del mismo, impidiendo el proceso de diseño no solo en una sola mente, sino también en la comunicación entre varias
- **Descubrimientos anteriores solo solucionaron dificultades accidentales**
    - **Lenguajes de alto nivel**
        - Permite abstraer gran parte de la complejidad no esencial (en particular aquello relacionado a la máquina / bajo nivel)
        - Sin embargo, no son capaces de proveer más que una comodidad para las entidades abstractas
    - **Time-sharing**
        - Mientras que no tan impactante como los lenguajes de alto nivel, la disminución del tiempo de respuesta del sistema y manteniendo la ilusión de feedback inmediato, permitiendo mejor concentración y un más consistente grasp de la complejidad del sistema
        - Sin embargo, nuevamente el tiempo de respuesta de un sistema se trata de una dificultad accidental
    - **Ambientes de programación unificados**
        - Unix e Interlisp
        - Atacan las dificultades accidentales del uso de programas juntos al proveer librerías integradas, formatos de archivos unificados, etc
        - Como resultado, estructuras conceptuales que en principio siempre pudieron usar la otra ahora pueden prácticamente hacerlo fácilmente
        - Esto facilitó el desarrollo de utilidades y herramientas
- **Esperanzas para la Silver Bullet**

- **Ada y otros avances en lenguajes de alto nivel**
  - Ada es un lenguaje de alto nivel de propósito general de los 80
  - Más importante que el lenguaje en sí es su filosofía, definida por modularización, tipos de datos abstractos, estructuración jerárquica
  - Sin embargo, se trata de otro lenguaje de programación más, y es probable que el principal impacto que termine teniendo sea no por una característica en particular, sino por el hecho de haber logrado que desarrolladores sean entrenados en técnicas de desarrollo de software modernas
- **Programación orientada a objetos**
  - Primero hay que distinguir entre dos ideas identificadas con el mismo nombre: tipos de datos abstractos y tipos de datos jerárquicos
    - El concepto del tipo de datos abstracto se refiere a que el tipo de un objeto debería ser dado por un nombre, un conjunto de valores apropiados y un conjunto de operaciones, en vez de su estructura de almacenamiento, la cual debería ser escondida
    - Los tipos jerárquicos permiten la definición de interfaces generales que pueden ser refinadas al proveer subtipos
  - Cada uno de estos tipos elimina una dificultad accidental del proceso, permitiendo al diseñador expresar la esencia de su diseño sin tener que considerar sintaxis innecesaria, permitiendo expresión de diseño de alto
  - Sin embargo, esto solo puede llegar a eliminar la dificultad en la expresión del diseño, y no la complejidad del diseño en sí
- **Inteligencia Artificial**
  - Dos diferentes definiciones en juego:
    - AI-1: El uso de computadoras para resolver problemas que antes podían ser solamente solucionados aplicando inteligencia humana
    - AI-2: El uso de un conjunto específico de técnicas de programación conocido como heurísticas. (Expert-systems technology dsp)
  - AI-1 solo ayuda a facilitar expresión, sin embargo, la complejidad principal está en decidir qué expresar, no expresarlo
- **Expert Systems**
  - Se tratan de programas que contienen un “inference engine” y una base de reglas, diseñadas para tomar inputs y suposiciones, y explorar las consecuencias lógicas a través de las inferencias derivables a partir de las reglas base, ofreciendo conclusiones y consejo, y la posibilidad de explicar su razonamiento al usuario
  - Una gran ventaja que tienen es que gran parte de los componentes modificables de la aplicación se encuentran en la base de reglas, regularizando gran parte de la complejidad de la aplicación
  - El poder de estos sistemas está dado principalmente por las cada vez más ricas bases de reglas que reflejan al mundo real más precisamente
  - Una de las dificultades más grandes para su uso es el prerequisito esencial de la necesidad de expertos que sepan el porqué y cómo de lo que hacen

    - † La contribución más importante de esta tecnología probablemente sea poner al servicio de programadores sin experiencia la sabiduría acumulada de los mejores programadores
  - **Programación automática**
    - † Generación de un programa para resolver un problema a partir de una especificación de la especificación del problema
    - † Critica que en la mayoría de los casos, es el método de la solución, y no el problema en sí, cuya especificación debe ser dada
    - † Mientras que admite que tiene usos específicos en los que es una herramienta poderosa, no cree que pueda llegar a poder generalizarse su uso, ya que dichos usos dependen de una caracterización dada por unos pocos parámetros y la existencia de múltiples soluciones
  - **Programación Gráfica**
    - † Básicamente vuelve al punto de antes, el software es inherentemente no visible
  - **Verificación de programas**
    - † Mientras que es un concepto muy poderoso, y puede llegar a encontrar un uso en casos particulares, como por ej. kernels seguros, las mismas son demasiado laboriosas para la industria
    - † La misma no asegura que no haya errores, ya que las pruebas en sí mismo podrían ser erróneas
    - † Lo más importante es que solo puede establecer que un programa cumpla con su especificación, cuando una de las partes más difíciles del desarrollo es llegar a una especificación consistente y completa
  - **Ambientes de desarrollo y otras utilidades**
    - † No espera mucho más allá de que ayuden en la productividad al evitar errores sintácticos y semánticos, así como una ayuda para manejarse en un sistema desarrollado por múltiples contribuidores
  - **Estaciones de trabajo (Las máquinas en sí)**
    - † La velocidad en su tiempo ya era aceptable, y mientras que compilación podría llegar a mejorar un poco más, las ganancias en productividad van a ser marginales
- **Ataques prometedores a la esencia conceptual**
  - Todos los ataques en la dificultad accidental están fundamentalmente limitados por la ecuación de la productividad: $\text{Time of task } = \sum (\text{Frequency})_i \times (\text{Time})_i$
  - Si los componentes conceptuales son los que toman la mayoría del tiempo, entonces debemos considerar aquellos ataques a las dificultades esenciales
  - **Desarrollar vs comprar**
    - † La solución más radical para construir software es no construirlo (xd?)
    - † Comprar es más barato que mantener a los programadores, y el delivery es inmediato
    - † El desarrollo del mercado en masa es por esto un gran posible contribuyente a la productividad del desarrollador

    - † Aplicabilidad es el problema principal, ya que hay que considerar si el producto sirve para el uso específico que se le desea dar
    - † Sin embargo, adaptarlo es un costo que se está dispuesto a dar, pues a diferencia de los 60 donde las máquinas eran mucho más inaccesibles y gastar un porcentaje de lo que se gastó en la misma en un software especializado no era nada, ahora las máquinas están mucho mejor distribuidas, el costo en ellas es mucho menor, y la adaptación de paquetes es mucho menos costosa que un programa especializado
        - ‡ La mejor manera de incrementar la productividad de un equipo intelectual sin conocimiento de la computadora es proveerlo con computadoras personales y programas generales de escritura, manejo de archivos, spreadsheet, etc
- **Refinamiento de requisitos y prototipado rápido**
    - † La parte más difícil del desarrollo es elegir precisamente qué construir, ninguna otra parte es más difícil de rectificar si hay errores
    - † Los clientes no saben lo que quieren, por lo que la función más importante que el desarrollador de software tiene para sus clientes es el proceso iterativo de ir refinando y entendiendo qué es lo que desean
    - † Por lo tanto, un ataque a la dificultad esencial es el desarrollo de tecnologías y técnicas que permiten el rápido prototipado de sistemas, simulando las interfaces y funcionalidades que no están en consideración en la iteración actual, permitiendo un ente concreto que el cliente puede testear
- **Desarrollo incremental - Crecer, no construir, software**
    - † Como las estructuras conceptuales son demasiado complicadas para ser especificadas correctamente, y demasiado complejas para ser construidas sin fallas, hay que tomar otro enfoque (dejar la metáfora de construir software)
    - † Sistemas deberían crecer de manera incrementativa, es decir, lograr que el sistema corra en un principio, incluso si esto implica que no haga algo útil y solo haga uso de “dummies”
    - † Luego, se va trabajando y creciendo este sistema, lo que lleva a un aumento impresionante en productividad, motivación, y capacidad de desarrollo de sistemas complejos
    - † **Grandes diseñadores**
        - ‡ Básicamente que son re importantes y quiere más plata

What is Software Design (Jack W. Reeves)

- Técnicas orientadas a objetos, y C++ parecen estar revolucionando la industria
- Parece que ya no se cuestiona si las técnicas son puro hype, y se empezó a preguntar cómo obtener sus beneficios con el mínimo sufrimiento necesario
- Porqué el interés repentino?
- Perspectiva Propuesta por el paper: C++ se volvió popular porque facilita diseñar software y programar a la vez
- Hay una lección que yace en la explosión en la popularidad de esta nueva tecnología: Programar no se trata de construir software, sino de diseñarlo (Parecido a silver bullet omg 🤯)

- No podemos ser ingenieros de software pues no sabemos qué significa realmente diseño de software
- El objetivo de cualquier actividad de ingeniería es llegar a algún tipo de documentación. Cuando un diseño es terminado, la documentación del mismo será entregada al equipo de manufacturación para proceder con la construcción, sin intervención de los diseñadores. Un ciclo de trabajo que en nuestra industria no ocurre salvo casos muy particulares
- El paper asume que el código fuente final es el verdadero diseño de software, y procede a analizar consecuencias de esa suposición
- La consecuencia principal de considerar al código como diseño de software es el hecho que el software es barato de construir, tanto que es casi gratis. Principalmente hecho por compiladores y linkers
- Otra consecuencia de considerar al código como diseño de software es que el mismo es fácil de crear, al menos en el sentido mecánico, permitiendo que crezcan rápidamente
- Con estas dos observaciones (Casi gratis de construir, y fáciles de realizar), tenemos que no es sorprendente que los diseños suelan ser increíblemente grandes y complejos
- Nota dos observaciones acerca de diseños de hardware:
  - † diseños de hardware complejos no están tan libres de errores como se suele creer
  - † Estos mismos diseños también vienen de la mano con costosos y complejos fases de construcción
- Esto último limita a aquellos que pueden manufacturar estos sistemas, límite que no aplica para el desarrollo de software
- Concluyendo que no parece que la industria tenga mucho que aprender de la de hardware
- El diseño de software es un ejercicio de gestión de complejidad, complejidad que yace en el diseño en sí, en la organización de software de la compañía, e incluso en la totalidad de la industria. (**Dificultad esencial?**)
  - † Abarca múltiples tecnologías y sub-disciplinas
  - † Especificaciones del software suelen ser fluidas y cambian constantemente
  - † Más similar a la complejidad de sistemas orgánicos que a la del desarrollo de hardware (**Había otro que lo comparaba así no?**)
- A comparación de los “verdaderos” ingenieros, quienes deben garantizar que sus diseños sean correctos y realmente sean buenos diseños antes de que comience el proceso de manufacturación, parecería que esto completamente elude al diseñador de software
- Sin embargo, el paper observa que esto no es del todo cierto, y que en realidad estamos constantemente validando y refinando nuestros modelos, solo que nosotros lo llamamos debugear y testear.
- La razón por la que no se consideran estas dos prácticas como una ingeniería parece ser un rechazo de considerar al código como diseño
- Desarrolladores de software no tienen necesidad de usar métodos más formales de diseño por el simple hecho de la facilidad económica
- Segunda revelación: Es más barato y más simple construir y testear el diseño que hacer cualquier otra cosa.

    - No cuesta prácticamente nada hacer infinitos builds del software
    - Aclara que testing es parte del refinamiento del diseño
    - Lo compara con ingenieros de hardware, quienes suelen construir modelos visuales para poder apreciar de mejor manera sus diseños.
    - Esta práctica es innecesaria para nosotros, simplemente podemos ir y construir el diseño en sí
- Critica el proceso de desarrollo de software que trata de acomodar a las distintas fases de manera estricta: El diseño de alto nivel debe ser completado antes de codear, y cualquier testeo / debugeo necesario es debido a errores en la etapa de construcción (Los programadores son los constructores de la industria)
    - Esta manera lleva a pensar a los programadores como los culpables de los errores que ocurren, lo que no es solamente erróneo, sino que también poco productivo
    - Hay que repensar la manera en la que organizamos
- El problema más prevalente en cuanto a diseño de software es el hecho de que todo es diseño: Codear, testear, debugear, e incluso lo que tradicionalmente se refiere con diseño, y todos estos aspectos están interrelacionados, por lo que no es realista esperar que se pueda eficientemente separar en etapas el proceso
- Software está intentando ser diseñado en capas, de manera que sea posible enfocarse en solo un módulo del diseño e ignorar el resto.
- Pero esto no es como realmente funciona, el diseño no está completo hasta que no sea construido y testeado, el diseño de alto nivel no es más que un framework estructural para el diseño detallado
- Es por esto que el diseño va a terminar influenciando el diseño de alto nivel. Este refinamiento es algo que debería ocurrir a lo largo del ciclo de diseño
- Software es demasiado complejo y sus dependencias son demasiado numerosas (Todo lo que puede llegar a salir mal, va a salir mal) como para que el proceso de diseño sea más riguroso
- No importa cuánto se trató de prevenirlos, errores siempre van a ocurrir durante el desarrollo. Esta es la diferencia entre la experiencia (craft) y engineering
    - Craft: La experiencia solo nos puede llevar hasta cierto punto dentro de territorio desconocido. Luego debemos empezar a mejorar lo que tenemos durante un proceso controlado de refinamiento, esto es engineering
- En la industria necesitamos desesperadamente buen diseño de todos los niveles, en particular de alto nivel, mientras mejor sea, más fácil va a ser el diseño final (el código). Cualquier herramienta que ayude a este objetivo está bien utilizarla, pero siempre hay que tener en mente que el verdadero diseño se hará con el código, por lo que no es mala idea codear nuestros diseños mientras los derivamos, refinando cuando sea necesario
- Sería mejor que los diseñadores originales también escriban el código, en vez de tener a alguien más que se encargue de la traducción. Lo que necesitamos es una notación adecuada para cada nivel del diseño. Acá es donde C++ aparece
- El lenguaje es apto ambos para proyectos y a la vez es un lenguaje de diseño de software más específico, permitiendo expresar directamente información de alto nivel acerca de distintos módulos, y facilitando el refinamiento luego, resultando en un diseño más robusto, mejor engineered

- Esta nueva tendencia en la industria muestra que instintivamente reconocemos que avances en técnicas de programación y lenguajes son más importantes que cualquier otra cosa. También demostrando que los programadores están interesados en el diseño
- También destaca como el proceso de desarrollo está cambiando (Cascada, spiral, etc), y que el principal beneficio que surgen de estas prácticas se debe al hecho de que se empieza a codear más temprano en el diseño de alto nivel, y no por alguna característica en particular del proceso
- Finalmente, destaca que el objetivo de cualquier proyecto debería ser producir algún tipo de documentación
  - † Es altamente probable que el sistema deba ser modificado luego
  - † Destaca dos importantes necesidades que deben ser resueltas con la documentación
  - † El primero es incluir la información del mundo real que no llegó a ser incluida en el diseño.
  - † El segundo es documentar aquellos aspectos difíciles de comprender a partir del diseño en sí. En general, herramientas visuales suelen ser claves en este paso, y reconoce la dificultad de mantener al tal documentación. Este es un argumento a favor de mantener la documentación minimal durante el refinamiento del diseño
  - † Real software runs on computers. It is a sequence of ones and zeros that is stored on some magnetic media. It is not a program listing in C++ (or any other programming language).
  - † A program listing is a document that represents a software design. Compilers and linkers actually build software designs.
  - † Real software is incredibly cheap to build, and getting cheaper all the time as computers get faster.
  - † Real software is incredibly expensive to design. This is true because software is

  incredibly complex and because practically all the steps of a software project are part of the design process.
  - † Programming is a design activity—a good software design process recognizes this and does not hesitate to code when coding makes sense.
  - † Coding actually makes sense more often than believed. Often the process of rendering the design in code will reveal oversights and the need for additional design effort. The earlier this occurs, the better the design will be

13 years later (noc si hara falta, pero debería leerlo por las dudas)

The Early history of Smalltalk (Alan C. Kay)

- Leer el abstract (Esto supongo debería hacerlo para todos)

Introducción

- Arranca destacando como el avance tecnológico de los 90 es lo que se tenía en mente durante los 60
- Smalltalk fue parte de una mayor búsqueda por parte de Xerox PARC y de ARPA, una búsqueda por personal computing
- Los lenguajes de programación pueden ser categorizados en una multitud de maneras, pero parecería ser que fundamentalmente o bien son una “aglutinación de features”, o bien son una “cristalización de estilo”. También destaca que los lenguajes presentes en la primer categoría suelen ser instigados por comités, mientras que aquellos pertenecientes a la segunda categoría, por una sola persona.

- El diseño de Smalltalk, y su existencia, se deben a la observación de que todo lo que podemos describir puede estar representado por la composición recursiva por un tipo particular de building block que esconde su combinación de estado y proceso dentro de sí mismo y con el que se puede interactuar solo mediante el intercambio de mensajes
  - Destaca la similitudes que los objetos de Smalltalk tienen con la corriente biológicas y físicas del siglo XX
  - Y el hecho que la manera de crear objetos es platónica en el sentido de que algunas actúan como idealizaciones de conceptos (Ideas), a partir de las cuales manifestaciones pueden ser creadas. Pero a su vez, las Ideas son manifestaciones, de la Ideas-idea y la idea-idea es un tipo de un tipo de manifestación-idea, que a su vez es un tipo de sí mismo. que el sistema sea completamente auto-descriptible - hecho que probablemente hubiese sido apreciado por Platón como una broma práctica
- Smalltalk es una recursión sobre la noción de computación en sí misma, en vez de dividar el dominio de la computadora en múltiples partes, cada objeto de Smalltalk es una recursión sobre todas las posibilidades de una computadora (Esto no lo entendí muy bien?) De esta manera, la semántica del lenguaje es como tener miles de computadoras inter-conectadas entre sí
- La contribución principal de Smalltalk es un nuevo paradigma de diseño - orientado a objetos - un intento exitoso en mejorar la eficiencia de modelar los cada vez más complejos sistemas dinámicos y relaciones de usuarios

1960-1966 Early OOP and other formative ideas of the sixties

- Dos motivaciones fueron centrales para OOP
  - La primera, a larga escala se buscaba un mejor esquema de módulos para sistemas complejos que involucre el ocultamiento de detalles
  - La segunda, a pequeña escala, era encontrar una versión más flexible de asignación, y luego removerla completamente
- Habla de etapas de aceptación de nuevas ideas (apenas notar un patrón -> notarlo pero no comprender su significado “Cósmico” -> usarlo operacionalmente en múltiples áreas -> volverse el centro de una nueva manera de pensar -> Ser parte del nuevo status quo del que inicialmente se separó
- Desde afuera, nuevas ideas van a ser criticadas como el producto de la locura, luego aceptadas como obvias y mundanas, para finalmenta ser reclamada como la invención de los que originalmente la criticaban
- Durante su laburo en las fuerzas aéreas, empezó la primera etapa al ver cómo un diseñador había logrado el pasaje de archivos entre sedes (En ese momento no había ni OS ni formatos de archivos estándar)
- También obtuvo algunas ideas acerca de traducción y evaluación de HLLs (High Level Languages) gracias al soporte innato de la B5000
  - Sin embargo, en el momento le pasó de largo otras de sus cualidades, como el modelo de memoria segmentado, sus mecanismos de protección y switching entre multiprocesos
- Se empezó a interesar en la simulación de sistemas, en particular de una máquina por otra
- Leyó el artículo de Moore
- **Sketchpad y Simula** (Videito)

- Llega a la universidad de Utah ARPA, donde aprende de Sketchpad y Simula, y relaciona a las dos y la manera en la que usan este esquema de maestros e instancias (Sketchpad) y actividades y procesos (Simula)
- Identifica a Simula como un lenguaje procedural para controlar "Sketchpad-like objects", con mucha más flexibilidad que los constraints de sketchpad
- Con esto relaciona todas los patrones que fue reconociendo en el sistema de archivos, la B5000, Sketchpad y Simula, dándole una epifanía: "Porque dividir el entero en partes más pequeñas? Porqué no en pequeñas computadoras, miles de ellas, con cada una simulando una estructura útil?"

1976-69 – The Flex Machine, A First Attempt At An OOP-Based Personal Computer

- Conoce a Ed, quien estaba trabajando en una computadora personal para no profesionales en computación, y buscaba programarla en un HLL. Kay sugiere JOSS
- Buscaban extender y simular dinámicamente, cosa de la que JOSS no era capaz, ya que era demasiado lento para computación seria
- Querían inspirarse / simplificar Simula (Era demasiado grande para su máquina de 16k palabras de 16-bits) para la Flex, no era completamente claro cómo
- Destaca que la belleza de JOSS yace la alta atención al usuario de su diseño
- Se inspira en el HLL EULER por ahora, pero destaca que sabía que el lenguaje FLEX debería estar semánticamente inspirado por Simula que por ALGOL o EULER, aunque no tenía claro cómo
- También estaba la duda de cómo usuario debería interactuar con la máquina

Doug Engelbart y NLS

- A principios de 1967, ARPA fue visitado por Doug, quien Kay llama "Un profeta de dimensiones bíblicas"
- Lo caracteriza como uno de los pioneros de lo que Kay empezó a denominar, con su trabajo en Flex, computación personal
- Presencio a Doug y su vision de los NLS (oNLineSystems) como el aumento del intelecto humano. su sistema era posta que profético, hyperlinks, cursor, gráficos, múltiples ventanas, ambiente colaborativo, entre otros
- Tuvo en cuenta mucha de estas ideas para la Flex
- En ARPA se busca la simbiosas computadora-humano?
- Aca empieza a considerar realmente la ley de Moore y sus implicaciones. Con precios disminuidos y con máquinas al alcance de cualquier usuario y no sólo los entrenados para hacer uso de las computadoras, necesariamente se iba a precisar de un sistema extensible en el que el usuario pudiera hacer el detallado final a sus herramientas
- La UI del NLS era prametrizable, pero la noción de de un menú jerárquico lo molestaba a Kay ya que esto implicaba qeue el usuario debería salir de un estado correspondiente al sistema antes de poder hacer cualquier trabajo. Quería considerar la posibilidad de implementar una interfaz mucho más plana
- La Flex machine:
  - Referencias a objetos fueron manejadas como una generalización de los descriptores de la B5000, con dos punteros. Uno para el objeto "maestro", y el segundo para la instancia del objeto

    - Para la asignación, := deja de ser un operador, y se vuelve en realidad una especie de índice que puede seleccionar un comportamiento determinado de un objeto. De esta manera, permite ver a todos los operandos relevantes a la vez
    - Los objetos son una especie de mapeo cuyos valores son su comportamiento
    - Los objetos deben ser dueños de manera privada de sus propios comportamientos
    - Como Simula, estructuras de control de corutinas era posible, y se hacía uso de cadenas de las mismas, conectadas mediante booleanos, para iterar. Idea que luego sería usada con mayor fuerza en Smalltalk
    - Revisar lo de **whens**
- Escucha por primera vez las nociones de que el sitema educativa está fallido y que la metodología de aprendizaje debería ser repensada
- Ve por primera vez a principios de 1968 un dispositivo de pantalla plana, y se cuestiona cuanto tardaría según Moore para que puedisese correr Flex
- Conoce el sistema GRAIL (Esencialmente un HLL con reconocimiento de escritura) y queda maravillado con su interfaz
- También se encuentra con un programa que tenía a niños programando, lo que le dió con otra epifanía que el destino de la computación personal sería un medio personal (no un mero vehículo personal que podía esperar hasta la adultez), y que por lo tanto debería extenderse al alcance de los niños también
- Todas estas ideas culminaron para darle una visión de lo que la computadora personal realmente debería ser (no más grande que un cuaderno, liviano, interfaz amistosa, y con el alcance de Simula y FLEX)
- Luego conoce el IMP system y su capacidad de extender la sintaxis del lenguaje, idea que le fascinó ya que la manera en que estaba implementado cada proceso definia su propia sintaxis, evitando una separacion de la parte de extension del resto del sistema
- Tambien aprendio profundamente LISP, y mientras estaba enamorado de la *idea*, estaba decepcionado de como a pesar de ser supuestamente un lenguaje funcional, sus componentes mas importantes: expresiones lambda, quotes, y conds, no eran funciones, sino algo llamado formas especiales
- “take the hardest and most profound thing you need to do, make it great, and then build every easier thing out of it.” That was the promise of LISP and the lure of lambda–needed was a better “hardest and most profound” thing. Objects should be it

1970-1972–XEROX PARC: The KIDDIKOMP, MINICOM, And SMALLTALK-71

- En 1970, Kay deja ARPA y se va a Palo Alto a unos laboratorios de computación de Xerox. Allí empieza a diseñar la KIDDIKOMP, que sería un prototipo para testear diseños de interfaces de computadoras personales
- A principios de 1971, varios intelectuales se unieron al laboratorio, PARC
- Querían tener acceso a la misma máquina en uso en ARPA por temas de continuidad, pero Xerox se niega. Terminan emulando la computadora ya que era más rápido que hacer un sistema operativo para la máquina que ya tenían
- Luego forma su equipo, Learning Research Group (LRG)

- A principios de 1971 refina el diseño de la KiddiKomp, ahora llamada miniCom, que usaba un lenguaje llamado Smalltalk. Nombre elegido para ir en contra de la corriente indo-europea de llamar a sus sistemas como dioses griegos, y porque el nombre le pareció tan inocuo que le entretuvo la idea que nadie se esperara algo del mismo
  - It was a kind of parser with object-attachment that executed tokens directly. (I think the awkward quoting conventions came from META.) I was less interested in programs as algebraic patterns than I was in a clear scheme that could handle a variety of styles of programming. The patterned front-end allowed simple extension, patterns as "data" to be retrieved, a simple way to attach behaviors to objects, and a rudimentary but clear expression of its eval in terms that I thought children could understand
- Seguía molesto que la belleza de Lisp fuese arruinada por sos componentes esenciales siendo en realidad "formas especiales". Quería pensar cómo objetos podían ser caracterizados como computadoras universales sin tener excepciones en la metáfora
- Sospechaba que lo que necesitaba era control completo sobre qué era pasado a través de mensajes, en particular, *cuando* y *en qué ambiente* las expresiones eran evaluadas
- Gracias a la tesis de Fisher acerca de la síntesis de estructuras de control, donde ofrece múltiples generalizaciones utilizadas en estructuras de sistemas de la época para simular una variedad de ambientes de control, entre otros. Así como soluciones al "funarg problem"
- Termina concluyendo (reflective design?) que lo necesario para hacer "OOP/Lisp de manera correcta" era manejar las mecánicas de invocación entre módulos sin preocuparse de los detalles de los modulos en sí.
- Quría asegurarse que el lenguaje fuera "bello", remarca como la naturaleza en sí es bella tanto en su elegancia y prácticamente - mencionando las membranas celulares y como forman un bloque de construcción y a la vez un punto de partida para la evolución
- Se le ocurre la idea de hacer que las ventanas puedan overlappearse, magnificando el área del display de la máquina, que era una de sus preocupaciones
- Estaba interesado en texto y presentaciones gráficas de alta calidad ya que supuso le facilitaría la posibilidad de usarlo como "caballo troyano" en las escuelas
- Luego de discusiones, empezó a preocuparse que el approach simbólico tomado por Smalltalk podría llegar a ser difícil para que los niños lo procesen, ya que la etapa simbólica recién comenzaba en sus vidas

1972-76—THE FIRST REAL SMALLTALK (-72), ITS BIRTH, APPLICATIONS, AND IMPROVEMENTS

- Kay se hace el piola y ahora tienen que demostrar que puede definir "el lenguaje más poderoso" en solo una página de código. Se le dificultó por tres razones:
  - Quería que fuera como un intérprete no recursivo
  - El entrelazamiento del parsing con la recepción de mensajes
  - Como el envío y la recepción debían interactuar entre sí
- Lo logra, y días después Dan Ingalls le dice "Acá tenés pa, corriendo y todo" (Lo codea en **Basic** el basado)
- Un tiempo después, consiguen la primer interim Dynabook, "Bilbo"

- 6 principios detras de smalltalk. Los primeros 3 son de “cerca de lo que son los objetos” – como se ven y son usados desde afuera y se mantuvieron sin cambios a lo largo de los años, mientras que los últimos tres –objetos desde adentro – fueron retocados varias veces
  1. Everything is an object
  2. Objects communicate by sending and receiving messages (in terms of objects)
  3. Objects have their own memory (in terms of objects)
  4. Every object is an instance of a class (which must be an object)
  5. The class holds the shared behavior for its instances (in the form of objects in a program list)
  6. To eval a program list, control is passed to the first object and the remainder is treated as its message
- LLegan a un estilo de encontrar comportamientos genéricos para símbolos de mensajes
- Como el control es pasado a la clase antes que el resto del mensaje sea considerado, la misma puede decidir no recibir el mensaje. Parte del ambiente es el bindeo del sender al mensaje
- Acá hay una demo de la sitaxis inicial y una clase que debería repasar.
- Queriá simplificar la sintaxis y hacerla mas plana y sin tantos paréntesis
- **Development of the Smalltalk-72 System and Applications**
  - El intérprete es lento pero usable
  - Tratan de lidiar con el problema de las overlapping windows. La clase Window fue la más modificada, al tener que considerar restricciones de hardware y avances en el área
    - † Usaron la convención de GRAIL de bordes sensitivos para poder mover, cambiar el tamaño, entre otras (básicamente el estándar de hoy xd)
    - † Window scheduling used a simple “loopless” control scheme that threaded all the windows together
  - Luego de las clases básicas, implementaron graficos tortuga, editor para código de Smalltalk dirigido por mouse, miniMOUSE (el primer editor WYSIWYG), soporte para documentos multimedia, la herramienta Findit, animaciones de objetos, efectos de sonidos musicales, PYGMELION (iconic programming?), una version simple de SIMULA llamada simpula
  - Que significa modeless?
- **The Evolution Of Smalltalk-72**
  - smalltalk-74 incorporó multiples mejoras tales como la existencia de un objeto “mensajero”, diccionarios de mensajes para las clases, bitblt (display), y una mejor y más general interfaz para las ventanas, y un sistema de memoria virtual (OOZE) permitiendo un mejor manejo de recursos y la posibilidad de recuperación de trabajo en caso de crasheos (que eran comunes en las primeras ALTOS)
  - También habla de varios de los mecanismos usados para éste propósito (repasarlo)
- **“Object-oriented” Style**
  - Diferencia la noción de su grupo de lo que era el estilo OOP y lo que denomina como una encapsulación artificial llamada “tipos de datos abstractos”. La generalización de estas últimas era tendencia en los 60, y se lo empezo a ver a Simula como un posible vehiculo para definirlas
  - what Simula had whispered was something much stronger than simply reimplementing a weak and ad hoc idea. What I got from Simula was that you could now replace bindings and assignment with goals. The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as sites of higher level behaviors more appropriate for use as dynamic components.

- Lamenta el hecho de mucho de a lo que se refiere con “programación orientada a objetos” en los 90 es en realidad el mismo viejo estilo de programación con estructuras fancy
- Pero entonces, de donde viene la eficiencia que asocia a OOP?
  - Una manera mucho mas clara de representar un systema complejo
  - El uso de cuatro tecnicas en harmonía:
    - Estado persistente
    - Polimorfismo
    - Instanciación
    - Métodos como objetivos para los objetos
  - Enfoca la mente del prograd or en una direccion particular
  - Tal vez el más importante, es el hecho de no darle privilegios ilimitados a cualquiera que tenga una estructura. Por esto todos los objetos son ciudadanos de primera y están protegidos
  - El hecho de que fuerza a un mejor diseño, el objeto conlleva consigo mucho significado e intención, llevando a código más compacto
  - Tough the late-binding of automatic storage allocation does not do anything a programmer cannot do, its presence leads to both simpler and more powerful code. OOP is a late binding strategy for many things and all of them together hold off fragility and size explosion much longer than the older methodologies. In other words, human programmers are not Turing machines–and the less their programming systems require Turing machine techniques the better.
- **Smalltalk And Children**
  - En 1973, buscan maneras de enseñarles a lis niños OOP
  - Un primer intento fue hacer uso de gráficos tortuga, pero no fue muy exitoso, por otro lado, lo que sí funcionó fue enseñandolo mediante ejemplos prácticos de programas completos
  - Observa que los éxitos que tuvieron en un principio no fueron tan generales como habían esperado, pero que en su momento estaban cegados por el hecho de que al menos algunos estuviesen realmente aprendiendo (“early succes syndrome” y “hacker phenomenon”)
  - Observaron que el problema que las mecánicas del sistema no parecían ser el problema, pero el diseño en sí
  - Termina realizando que el problema es que no saben diseñar, y las soluciones que él se esperaba fueran simples de realizar eran imposibles sin conocer las ideas de antemano (Lo compara con la literatura y el lenguaje), y menciona el hecho de que tener ideas más poderosas ayuda a prender nuevas de mejor manera
  - Con esto en mente decidieron empezar a enseñar diseño a los niños, para lo que harían uso de un design template
  - Sin embargo, esto seguía sin ser suficiente, trataron de probar con inheritance, pero resultó un concepto muy complejo
  - Se pregunta si el problema no era tanto un error en como encaraban los experimentos, pero mas bien el hecho que para que la mayoría de los niños sea capaz de pensar como un programador se necesita tiempo (lo compara con escribir)
  - Con esto en mente el problema se vuelve no lograr que los niños hagan algo, que es algo que por default les encanta hacer, sino qué ideas pririzar y poner mas énfasis y que tanto dejar que penetran el desarrollo mental del niño, problemas que incluso otros campos mucho más antiguos tienen (matemática, lenguaje, escritura)

† Leer el último análisis que justifica el porqué enseñar programación (Se parece bastante a la idea de la teoría de Ryce?)

1976-1980–THE FIRST MODERN SMALLTALK (-76), ITS BIRTH, APPLICATIONS, AND IMPROVEMENTS

- A fines de 1975 sentía que estaban perdiendo el balance, la "dynabook para niños" se estaba esfumando, por lo que decidieron irse durante tres días a otro lado a mencionar todos los problemas
- Kay quería empezar desde cero, un nuevo sistema hardware-software diferente a la ALTO y Smalltalk. Todos estaban en acuerdo que el poder de Smalltalk no alcanzaba las aspiraciones del grupo. Hubo una parcial fractura en el grupo ya que Kay no creía que OOP por sí mismo iba a solucionar los problemas con los end-users, mientras que otros querían un Smalltalk más rápido y que pudiera solucionar problemas más grandes.
- Dan Ingall empezó a trabajar en Smalltalk 76, mientras que Kay se enfocó en una nueva máquina pequeña y un nuevo lenguaje llamado NoteTaker
- Kay estaba preoupado que Smalltalk y OOP lo hubieran formado y sesgado en sus pensamientos, reconoció que veía en Smalltalk algo impresionante, pero no el lenguaje para el end-user ni una solución de un medium de escritura y lectura para los niños
- Dan Ingalls siguió con smalltalk, lo primero que solucionó fue la dualidad clase / función en favor de una definición completamente intensional donde cada pieza de código era un método intrínsico. Desambiguó el lenguaje, sacándole extensionabilididad y flexibilidad al mismo (activation records ahora tenían que ser objetos?), cosa que todo el grupo estaba de acuerdo estaba en exceso en las otras versiones de Smalltalk, logrando un compilador e intérprete eficiente capaz de correr 180 veces mas rápido
- **Inheritance** (releer)
  - Debido a la flexibilidad y extensibilidad de Smalltalk 72 y al hecho que es un lenguaje dinámico, decidieron dejar afuera inheritance como feature, ya que la misma podía ser simulada
  - Para Smalltalk 76, Dan Ingalls utilizó un esquema similar a Simula en cuantro a semántica, pero que podía ser cambiado incrementalmente para poder alinerse con los objetivos del grupo de estrecha interacción (?), y también incluyo múltipke herencia, pero no era limpio
  - Kay no estaba completamente convencido, ya que creía que necesitaban una nueva teoría acerca de herencia (lo sigue creyendo). Da como ejemplo instanciación y herencia (no entendi)
  - Trata de convencer a Xerox que el futuro está en la computación personal, pero falla
- **The Smalltalk User Interface**
  - Muchas de los elementos utilizados ya estaban en existencia en los 60, pero el gran cambio que consolidó estas ideas en una teoría poderosa vino del énfasis de LRG en niños
  - Tenían un énfasis en aprendizaje como uno de los objetivos principales, generando un cambio del propósito de la interfaz de usuario de "acceso a funcionalidad" a "ambiente en el que los usuarios aprenden haciendo"
  - El objetivo principal de LRG era encontrar el equivalente a escribir – aprender y escribir haciendo cosas en un medium – por lo que se conformaron con "iconic programming"

- En resumen, buscaban un ambiente en el que exploración cause sequencias deseadas, que permita prendizaje icónico y simbólico, que se sienta familiar, y que actúe como un espejo de aumento para el intelecto del usuario

- **Smalltalk 76**
  - El nuevo diseño e implementación era eficiente y divertido. Tenía las funcionalidades de OS, Ethernet, archivos, impresión, interfaz de ventanas, editores, gráficos, y dos nuevas contribuciones, un browser para métodos estáticos en la cadena de herencia y contextos dinámicos para el ambiente del debugger
  - En 1978 el sistema tuvo su primera prueba cuando iba a ser probado por múltiples ejecutivos de Xerox
  - Para la situación speedrunearon un ambiente de aprendizaje autocontenido hecho para adultos sin experiencia (De acá sacaron la idea de hacer el tamaño de la fuente configurable)
  - No entendí la parte de pushing y pulling
  - Durante 1978, Kay siguió trabajando con Notetaker, pero terminó resultando que los chips que estaba esperando empezacen a ser manufacturados no lo serían, por lo que tuvieron que satisfacerse con Intel 8086
  - Dan se interesó en Notetaker y quería ver si podía hacer una versión de Smalltalk-76 que pudiera ser el sistema de la maquina. Con solo 256K de memoria y solo floppy, el diseño del sistema mejoró bastante. La tabla de objetos indexable surgió como una manera de simplificar acceso a objetos, y el kernel total de la máquina fue reducido a 6K bytes de codigo del 8086 (no le gusta ***NADA*** a kay el chip)
  - El intérprete era el doble de rápido que la ALTO, principalmente porque la ALTO no tenía suficiente memoria de microcódigo para todo el código de emulación y por lo tanto el mismo era emulado en código de NOVA, forzando dos capas de interpretación
  - Lamentalblkemente, Xerox seguía sin apoyarlos para llevarlas al mercado
  - En 1979 hicieron la demo a Steve Jobs y un grupo de Apple, demostrando en vivo el poder de un sistema incremental
  - Apple trató de comprar la tecnología pero Xerox no los dejó, y tampocó decidió hacer algo con lo que tenían

1980-1983–THE RELEASE VERSION OF SMALLTALK (-80)

- Dan estaba motivado a liberar Smalltalk después del rechazo de la Notetaker, pero Kay no. Ya estaba satisfecho con el diseño de Smalltalk, y triste que su sueño de llevarla a los niños nunca se realizó completamente.
- Algunos miembros fueron contratados por Apple para trabajar en Lisa, Kay tomó un año sabatico
- Los que se quedaron decidieron que tenían que adaptar Smalltalk para poder ser usado, independientemente del hardware, empezando por cambiar las fuentes custom por las estándares ASCII.
  - Se ásemejaron a los bloques a las expresiones lambda, y se agregaron metaclases, a confusion de Kay
  - Habla de que pudieran haber implementado un “lenguaje observador”, pero supuso no lo hicieron porque ya estaban en la etapa en el que una manera de hacer las cosas se convertía en un dogma

- **Coda**
  - Hardware es simplemente software cristalizado tempranamente
  - Habla acerca de cómo gran parte del desarrollo de software es bindear lo más tarde posible, e hincharle las bolas a los manufacturadores para que agreguen optimizaciones necesarias al hradware (**leer esta seccion bien**)
  - Hace un comentario acerca de como tiene una mini teoría de que grandes cambios en lenguajes de programación ocurren cada 11 años, pero en los 80 parece no haber habido innovación
  - Culpa de esto a enfocar a las mentes en aplicaciones principalmente prácticas, y la aversidad de las compañías a hacer su propio hardware, habla de un paso atrás en todos los aspectos
  - Recalca que un problema del siglo 20 era que las tecnoloias se volvieron demasiado fáciles, por lo que es más fácil no tomarse el tiempo necesario para hacer las cosas bien (Esto se parece al punto que hacen en What is software design acerca de industria del hardqare?)
  - Cierra preguntandose donde estan los Dan Ingalls y Adeles de los 80 y 90