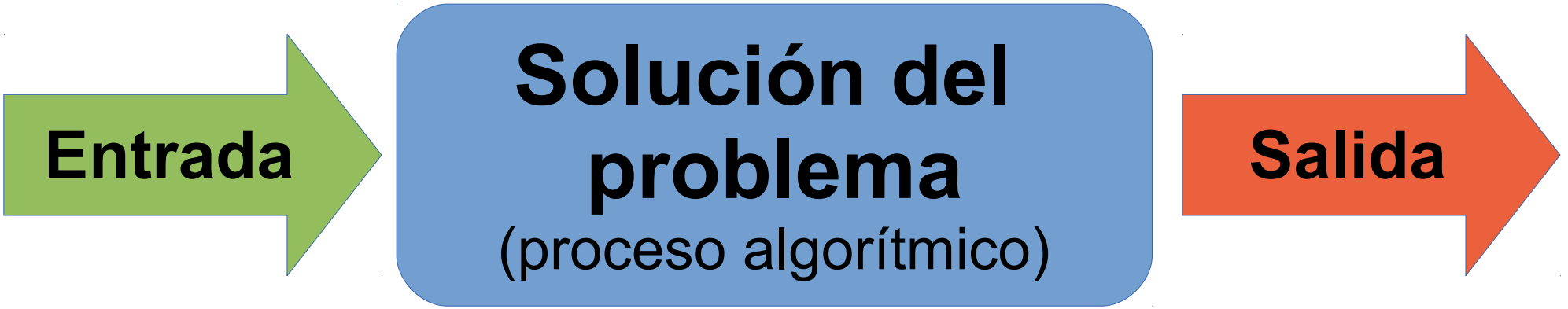
Proceso de solución de problemas

José Galaviz

Entender el problema

- ¿Qué es lo que queremos obtener?
- ¿Cuales son los datos que tenemos para obtenerlo?
- ¿Son suficientes?
- ¿Qué hace que el resultado obtenido resuelva el problema? ¿Cual es la característica que hace de un resultado, una solución?
- ¿Qué operaciones o construcciones se deben obtener para llegar a la solución?

La solución es...



Entrada

- Tipo
- Formato
- Tamaño
- Cantidad
- Fuente

Salida

- Tipo
- Formato
- Tamaño
- Cantidad
- Destino

Requisitos

Funcionales

(qué debe hacer el programa)

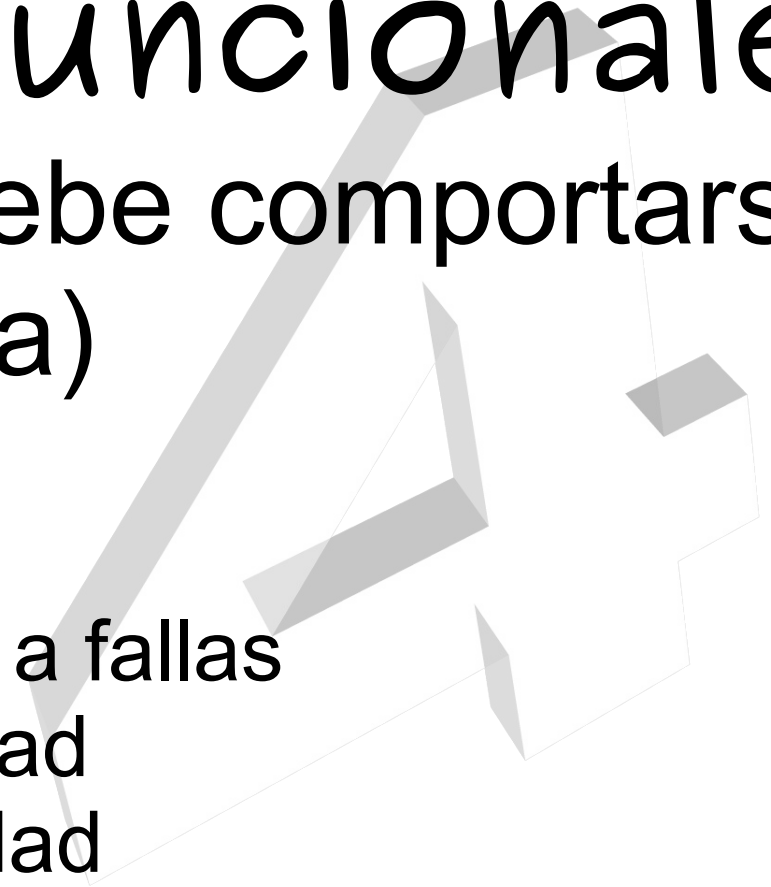
- Qué debe entregar como salida dado cierto tipo de entrada.
- Qué se debe llevar a cabo.

Requisitos

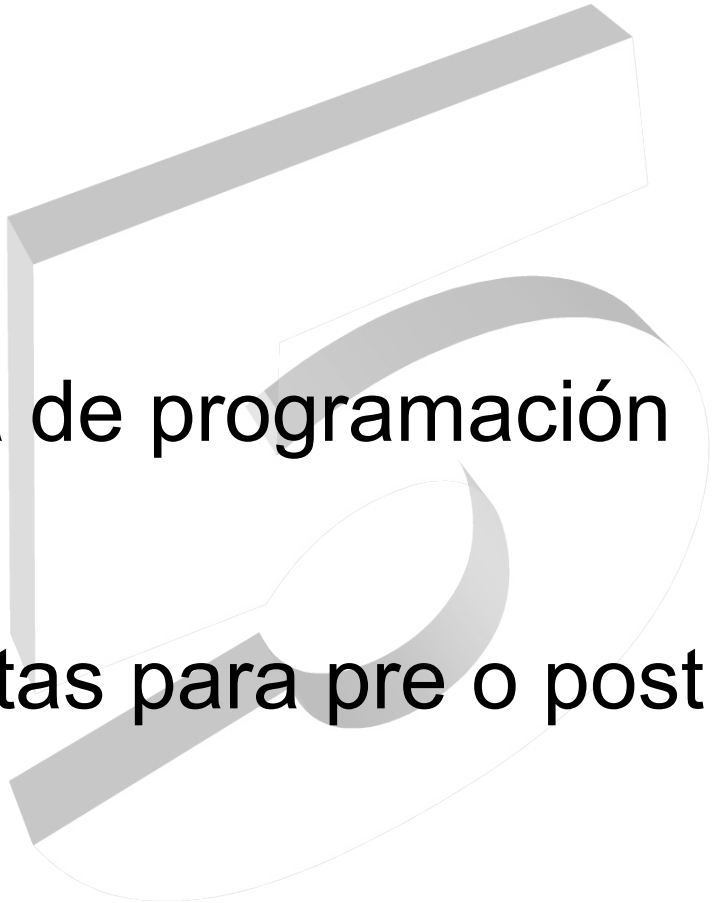
No-funcionales

(cómo debe comportarse el programa)

- Eficiencia
- Tolerancia a fallas
- Amigabilidad
- Escalabilidad
- Seguridad
- Interoperabilidad
- ...



Elegir parte del arsenal



- Paradigma de programación
- Lenguaje
- Herramientas para pre o post procesar los datos

Escatimar trabajo inútil

- ¿Este problema o alguno similar ya ha sido resuelto antes?
- ¿Cómo?
- ¿Se puede reutilizar algo de lo ya hecho?
- ¿En qué es diferente este problema de otros similares que se han resuelto?

Esbozar el proceso de solución (enfoque procedural)

- Definir, *grosso modo*, los pasos que se deben llevar a cabo
- Qué datos entran en cada paso
- Qué datos salen de cada paso

Esbozar el proceso de solución (enfoque orientado a objetos)

- Definir, las clases de objetos involucrados en el problema.
- Definir sus atributos
- Definir su comportamiento

Elegir el modelo de datos

- Cómo se representarán los datos
- Qué estructuras de datos son las adecuadas para almacenarlos
- De qué manera la estructura de datos determina el desempeño
- El modelo elegido ¿se ajusta a los requisitos funcionales?, ¿y al esbozo general de solución?
- ¿Podría haber algo mejor?
- ¿Podría haber algo peor?, ¿por qué es peor?

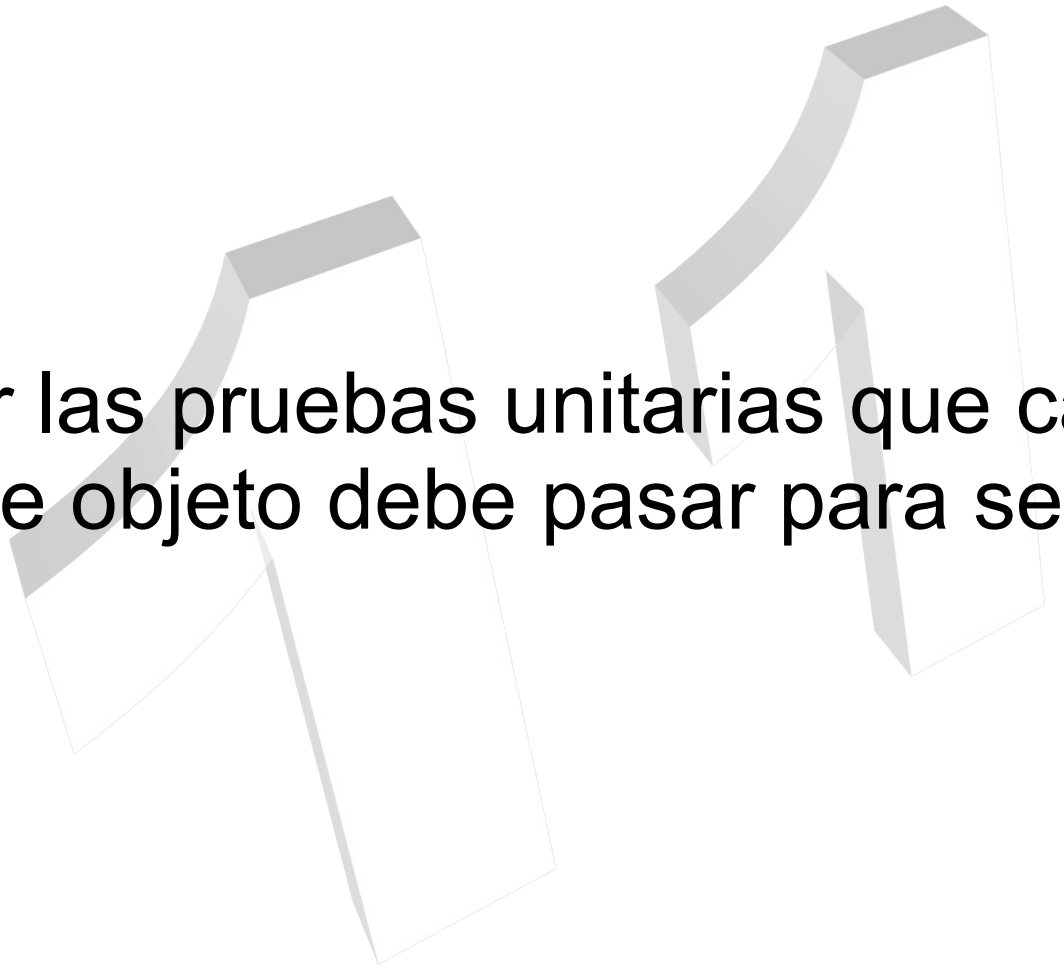
El resto del arsenal

- ¿Hay frameworks o bibliotecas que puedan utilizarse?
- ¿Qué tan útiles son?
- ¿Qué se necesita para usarlas?
- ¿Están bien documentadas?
- ¿Han sido probadas?
- ¿Vale la pena usarlas? (costo-beneficio)
- ¿Son seguras?

Descomposición

- En OO: definir clases y objetos.
- En programación estructurada: definir sub-procesos o subrutinas
- Asignar responsabilidades
- Definir servicios e interfaces para usarlos
- Esclarecer qué se necesita para proveerlos
- Alta cohesión
- Bajo acoplamiento
- Encapsulación

Pruebas



- Definir las pruebas unitarias que cada módulo o tipo de objeto debe pasar para ser confiable

Implementación (enfoque procedural)

- Refinando cada vez más la descripción hecha del proceso.
- *Top – Down*
- De lo general a lo particular. De lo poco detallado a lo muy detallado. De lo abstracto a lo concreto

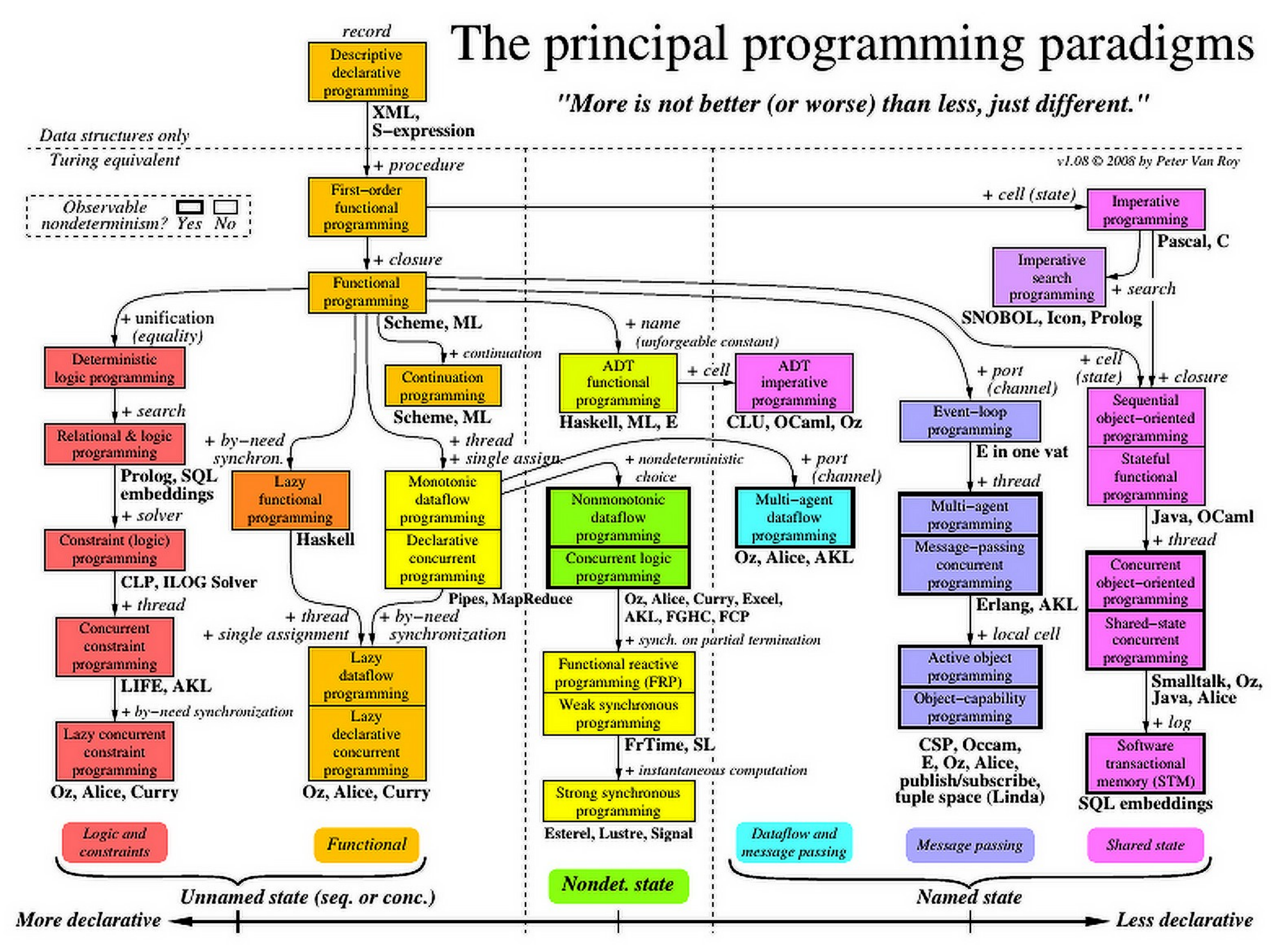
Implementación (enfoque orientado a objetos)

- Desarrollar el sistema partiendo de lo más elemental
- *Bottom - Up*
- De lo particular a lo general. Construyendo servicios sofisticados y complejos sobre los más primitivos y simples ya elaborados. De lo concreto a lo abstracto

Verificación y pruebas globales

Depuración

Paradigmas diferentes


The principal programming paradigms
"More is not better (or worse) than less, just different."
v1.08 © 2008 by Peter Van Roy
record
Descriptive declarative programming
XML, S-expression
Data structures only
Turing equivalent
+ procedure
First-order functional programming
Observable nondeterminism? Yes No
+ closure
Functional programming
Scheme, ML
+ cell (state)
Imperative programming
Pascal, C
Imperative search programming
+ search
SNOBOL, Icon, Prolog
+ unification (equality)
Deterministic logic programming
+ search
Relational & logic programming
Prolog, SQL embeddings
+ solver
Constraint (logic) programming
CLP, ILOG Solver
+ thread
Concurrent constraint programming
LIFE, AKL
+ by-need synchronization
Lazy concurrent constraint programming
Oz, Alice, Curry
+ continuation
Continuation programming
Scheme, ML
+ name (unforgeable constant)
ADT functional programming
Haskell, ML, E
+ cell
ADT imperative programming
CLU, OCaml, Oz
+ by-need synchron.
Lazy functional programming
Haskell
+ thread
+ single assign.
Monotonic dataflow programming
Declarative concurrent programming
Pipes, MapReduce
+ nondeterministic choice
Nonmonotonic dataflow programming
Concurrent logic programming
Oz, Alice, Curry, Excel, AKL, FGHC, FCP
+ port (channel)
Multi-agent dataflow programming
Oz, Alice, AKL
+ thread
+ single assignment
+ by-need synchronization
Lazy dataflow programming
Lazy declarative concurrent programming
Oz, Alice, Curry
+ synch. on partial termination
Functional reactive programming (FRP)
Weak synchronous programming
FrTime, SL
+ instantaneous computation
Strong synchronous programming
Esterel, Lustre, Signal
+ port (channel)
Event-loop programming
E in one vat
+ thread
Multi-agent programming
Message-passing concurrent programming
Erlang, AKL
+ local cell
Active object programming
Object-capability programming
CSP, Occam, E, Oz, Alice, publish/subscribe, tuple space (Linda)
+ cell (state)
+ closure
Sequential object-oriented programming
Stateful functional programming
Java, OCaml
+ thread
Concurrent object-oriented programming
Shared-state concurrent programming
Smalltalk, Oz, Java, Alice
+ log
Software transactional memory (STM)
SQL embeddings
Logic and constraints
Functional
Dataflow and message passing
Message passing
Shared state
Unnamed state (seq. or conc.)
Nondet. state
Named state
More declarative
Less declarative

Programación Procedural

Ante el problema, concebido como proceso

- Analizarlo para descomponerlo en sub-procesos más simples.
- Refinándolos iterativamente hasta poseer unidades de complejidad manejable susceptibles de ser programadas, probadas y depuradas.
- Top-down, process-oriented.

Programación Orientada a Objetos

Ante el problema, concebido como proceso

- Analizarlo para determinar los actores en él involucrados, sus relaciones, las acciones que llevan a cabo y los servicios que proporcionan.
- Obtener clases de objetos de complejidad manejable, cuyos servicios son simples de implementar *per se* o basándose en los de otros.
- Bottom-up.

Diferencia de enfoque

- PP: ¿Qué se debe hacer? ¿Qué se necesita saber para hacerlo?
- OO: ¿Qué tipo de cosas se deben tener? ¿Qué pueden hacer estas cosas para resolver el problema?

Complejidad

- Ambos paradigmas pretenden simplificar. Hacer manejable la complejidad.

¿Cual suena mejor?

- Para un computólogo suena más razonable el procedural.
- Ejemplo. Pensemos en hacer una hoja de cálculo.

¿Te cae?

- Ahora, pensemos en hacer un sistema manejador de ventanas.

Software

- Software más amigable con el usuario.
- Con cualquier usuario.
- Los usuarios de hoy día son mucho más demandantes que los de la década de los 70.
- Sus necesidades se traducen en software más complejo.
- Que sería impensable hacer sin objetos.

Vamos a usar ambos

- ¿Por qué?
- Porque la programación estructurada vive **dentro** de la orientada objetos.
- Porque hay problemas más complejos que otros.
- Los simples se resuelven más eficientemente con estructurada.