v1.0
20140316

Programmazione Avanzata e Paradigmi

Ingegneria e Scienze Informatiche - UNIBO
a.a 2013/2014
Lecturer: Alessandro Ricci

[module 1.1]
PROGRAMMING PARADIGMS: OVERVIEW

SUMMARY

- What’s a programming paradigm
  - basic terms
- Main programming paradigms
  - imperative, functional programming, logic programming, object-oriented programming
  - multi-paradigm programming
- Taxonomy by Van Roy
  - observable non determinism, state
  - creative extension principle

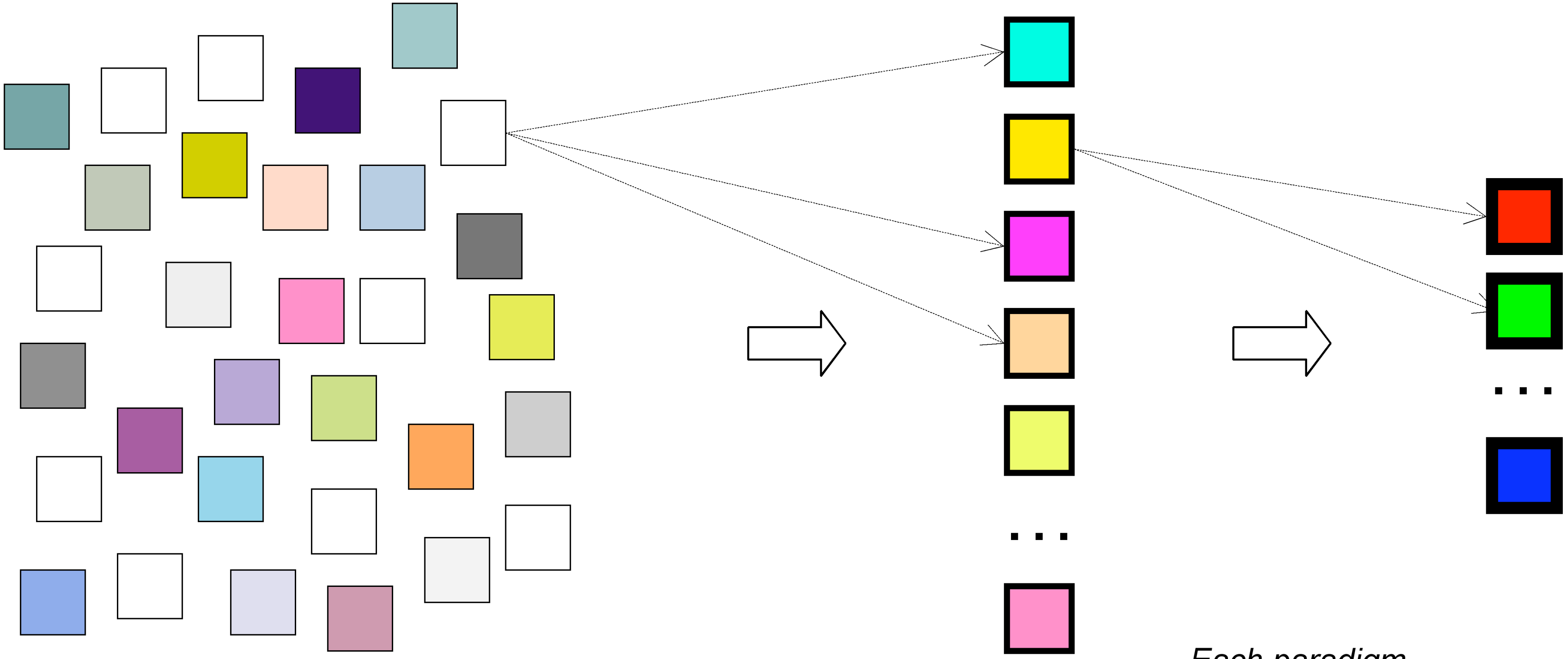
WHAT’S A PARADIGM

- The Merriam-Webster's Collegiate dictionary:
  - *“A <u>philosophical</u> and <u>theoretical framework</u> of a scientific school or discipline within which <u>theories</u>, <u>laws</u>, and <u>generalizations</u> and the <u>experiments</u> performed in support of them are formulated"*
- ***Programming* paradigm:**
  - *A programming paradigm is an approach to programming a computer based on a mathematical theory or a coherent set of principles* (Van Roy, CTM)
    - each paradigm supports a set of concepts that makes it the best for a certain kind of problem.
  - *A programming paradigm is a fundamental style of computer programming* (Wikipedia, March 2013)
  - *A pattern that serves as a school of thoughts for programming of computers* (Kurt Nørmark, Aalborg University, Denmark)

A PROGRAMMING PARADIGM IS

- ...how *computation is expressed* and works
- ...how a *program is organized* (program design perspective)
  - **structure** - what parts
  - **behaviour** - how parts compute
  - **interaction** - how parts interact

PARADIGMS & LANGUAGES



Each language realizes one or more paradigms

Each paradigm consists of a set of concepts

Languages ⟶ Paradigms ⟶ Concepts

PARADIGMS & ELEMENTS OF PROGRAMMING

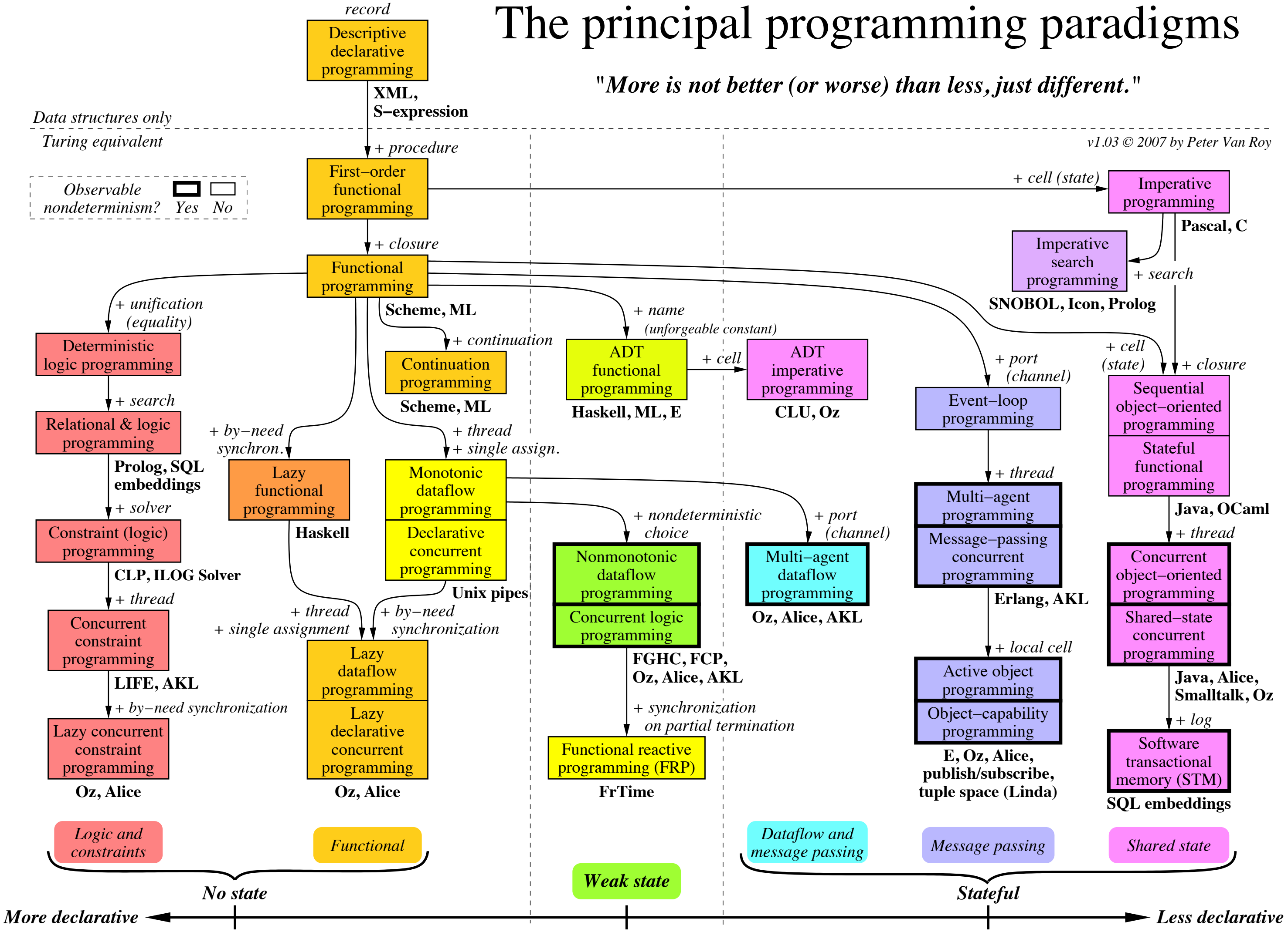
- Programming languages *as frameworks within which we organise our ideas about processes*
- 3 main mechanisms:
  - ***primitive*** *expressions,* which represent the simplest entities the language is concerned with
  - *means of* ***combination***, by which compound element are built from the simpler ones
  - *means of* ***abstraction***, by which compound elements can be named and manipulated as units

\> a paradigm typically defines specific concepts and mechanisms for these three dimensions

SEVERAL PARADIGMS

(...AND EVEN MORE LANGUAGES)

- Van Roy’s Taxonomy preview



The principal programming paradigms

"More is not better (or worse) than less, just different."

v1.03 © 2007 by Peter Van Roy

Data structures only

Turing equivalent

Observable nondeterminism? *Yes* *No*

record

Descriptive declarative programming

XML, S−expression

+ procedure

First−order functional programming

+ cell (state)

Imperative programming

Pascal, C

+ search

Imperative search programming

SNOBOL, Icon, Prolog

+ closure

Functional programming

Scheme, ML

+ 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

+ continuation

Continuation programming

Scheme, ML

+ by−need synchron.

Lazy functional programming

Haskell

+ thread
+ single assign.

Monotonic dataflow programming

Declarative concurrent programming

Unix pipes

+ thread
+ single assignment

+ by−need synchronization

Lazy dataflow programming

Lazy declarative concurrent programming

Oz, Alice

+ name (unforgeable constant)

ADT functional programming

Haskell, ML, E

+ cell

ADT imperative programming

CLU, Oz

+ nondeterministic choice

Nonmonotonic dataflow programming

Concurrent logic programming

FGHC, FCP, Oz, Alice, AKL

+ synchronization on partial termination

Functional reactive programming (FRP)

FrTime

+ port (channel)

Multi−agent dataflow programming

Oz, Alice, AKL

+ port (channel)

Event−loop programming

+ thread

Multi−agent programming

Message−passing concurrent programming

Erlang, AKL

+ local cell

Active object programming

Object−capability programming

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

Java, Alice, Smalltalk, Oz

+ log

Software transactional memory (STM)

SQL embeddings

Logic and constraints

Functional

Dataflow and message passing

Message passing

Shared state

Weak state

No state

Stateful

More declarative ← → ***Less declarative***

A LOOK TO HISTORY

1936 - Turing Machine
1936 - Untyped Lambda Calculus by Alonzo Church
1940 - Typed Lambda Calculus by Alonzo Church
1945 - Von Neumann Architecture
1949 - EDSAC computer, has an assembly language
1957 - FORTRAN (First compiler)
1958 - LISP
1958 - ALGOL 58
1959 - COBOL
1961 - MULTI-PROGRAMMING & TIME-SHARING OS (OS, INTERRUPT)
1962 - APL
1962 - Simula
1964 - BASIC
1965 - Dijkstra - Cooperating Seq. Processes + Semaphores
1968 - Logo
1970 - FIRST DEVELOPMENT OF UNIX OS

A LOOK TO HISTORY

1970 - Pascal
1971 - Monitors
1972 - C
1972 - Smalltalk
1972 - Prolog
1973 - ACTOR MODEL
1973 - ML
1974 - Internet protocol
1975 - Scheme
1975 - Concurrent Pascal
1978 - SQL
1978 - Hoare introduces CSP

A LOOK TO HISTORY

1980 - C++ (as C with classes, name changed in July 1983)
1980 - CCS - Calculus of Communicating Processes (Milner)
1982 - TCP/IP
1983 - Ada
1984 - Common Lisp
1984 - MATLAB
1985 - Eiffel
1986 - Objective-C
1986 - Erlang
1988 - Mathematica

A LOOK TO HISTORY

1990 - Haskell
1991 - Python
1991 - Visual Basic
1991 - Web & HTML (Mark-up Language)
1993 - pi-calculus
1993 - Ruby
1993 - Lua
1993 - Newton message pad
1994 - CLOS (part of ANSI Common Lisp)
1995 - Java
1995 - JavaScript
1995 - PHP
1998 - Google

A LOOK TO HISTORY

2001 - C#
2001 - Visual Basic .NET
2002 - F#
2004 - IBM X10
2005 - Multi-core era / “the free lunch is over” begins
2007 - mobile with smart phone / mobile app begins (iPhone, Android)
2007 - Clojure
2009 - Go
2010 - mobile with tablets
2011 - Dart
2012 - Typescript

MAIN PROGRAMMING PARADIGMS

- Four main paradigms
  - the **imperative** paradigm
  - the **functional** paradigm
  - the **logical** paradigm
  - the **object-oriented** paradigm

IMPERATIVE PARADIGM

"First do this and next do that"

- Describes computation in terms of *statements that change a program state*
- Imperative programs define *sequences* of statements *or commands* for the computer to perform
  - command => measurable effect on the program state
  - the order to the commands is important
- Representative languages
  - Fortran, Algol, Pascal, Basic, C

IMPERATIVE PARADIGM

- Origin/inspiration
  - digital hardware technology and the ideas of Von Neumann
- Reference computation model
  - Turing Machine

IMPERATIVE PARADIGM

- Incremental *change of the program stat*e as a function of *time*
- Execution of computational steps in an order, governed by control structures
- Computational steps referred as (synonyms):
  - "*statement*" - often used to refer to an elementary instruction in a source language
  - "*instruction"* - to be preferred to explicitly refer to the computational steps performed at the machine level.
  - "*command" -* often used to refer to actions in imperative programming language
    - e.g. assignment, IO, procedure calls

IMPERATIVE PARADIGM

```
n := x;
a := 1;
while n > 0 do
begin
  a := a * n;
  n := n - 1
end;
```

IMPERATIVE PARADIGM - ABSTRACTIONS

- The natural abstraction is the **procedure**
  - abstracts one or more actions to a procedure, which can be called as a single action
- Procedural programming
  - programs as collection of procedures
  - state changes are *localized to procedures* or restricted to explicit arguments and returns from procedures
- Structured, **modular** programming
  - fundamental for the maintainability and overall quality of imperative programs
    - OOP is the next step

FUNCTIONAL PROGRAMMING

"Evaluate an expression and use the resulting value for something"

- Computation is carried on entirely through *the evaluation of expressions*
  - represented by *functions* without side effects
- no state, no mutable data
- Representative languages
  - Haskell, F#, Erlang, ML, Scheme, Lisp

FUNCTIONAL PROGRAMMING

- Origin and inspiration
  - mathematics and the theory of functions
- Reference computation model
  - lambda calculus (λ-calculus)

FUNCTIONAL PROGRAMMING

```
fac 0 = 1
fac n = n*fac(n-1)
```

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

```
> map fac [2,5,3]
[2, 120, 6]
```

FUNCTIONAL PROGRAMMING - ABSTRACTION

- The natural abstraction is the **function**
  - abstracts a single expression to a function which can be evaluated as an expression
- Functions are first class values
  - functions are typed data just like numbers, lists, ...
  - can be passed as arguments to other function
    - *high-order functions*
- Applicative
  - all computations are done by applying (calling) functions
  - the values produced are non-mutable
  - no loops, recursion!

LOGIC PROGRAMMING

> *"Answer a question via search for a solution"*

- Programs consist of logical statements, and the *program executes by searching for proofs* of the statements
- Particularly effective for problem domains dealing with the extraction of knowledge from basic facts and relations
  - AI domain
- Representative languages
  - Prolog, Datalog

LOGIC PROGRAMMING

- Origins and inspiration
  - automatic proofs within artificial intelligence
- Reference computation model
  - first-order logic

LOGIC PROGRAMMING

```
female(anna).
female(elettra).
male(vinicio).
parent(vinicio,anna).
parent(elettra,anna).
son(X,Y) :- male(X), parent(Y,X).
daughter(X,Y) :- female(X), parent(Y,X).
```

```
append([],L,L).
append([X|L1],L2,[X|L3]) :-
  append(L1,L2,L3).
```

```
fac(0,1).
fac(N,F) :-
  N1 is N-1, fac(N1, F1), F is N*F1.
```

LOGIC PROGRAMMING - ABSTRACTIONS

- Based on axioms, inference rules, and queries
- Program execution becomes a systematic search in a set of facts making use of a set of inference rules
- Algorithms = Logic + Control
  - programs must specify only the logic side
  - the control side is totally handled by the abstract machine

DECLARATIVE PROGRAMMING

Functional Programming

Logic Programming

Expresses the logic of a computation without explicitly describing a control flow

OBJECT-ORIENTED PROGRAMMING

"*Send messages between objects to simulate the temporal evolution of a set of real world phenomena*"

- Computation given by the *exchange of messages* among self-contained computational objects with an identity and state
  - encapsulating a state and a behavior
- Strong support of encapsulation
  - key issues when programs become larger and larger.
- Conceptual anchoring of the paradigm to problem domains
  - objects represent concept of the problem domain

OBJECT-ORIENTED PROGRAMMING

- Origins and inspirations
  - the theory of concepts, and models of human interaction with real world phenomena
- Representative Languages:
  - Smalltalk/Squeak, C++, Java, Objective-C, C#, Scala, Python, Ruby,...

OOP ROOTS

- Modeling and discrete-event simulations
  - Simula language (1960s)
    - Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in Oslo
- Smalltalk
  - Alan Kay and his group at Xerox PARC (1970s)
    - introduced the term object-oriented programming = use of objects and messages as the basis for computation
  - BYTE Special Issue on Smalltalk and OOP - August 1981

OBJECT-ORIENTED PROGRAMMING - SOME KEY CHARACTERISTICS

- **Encapsulation**
  - data as well as operations are encapsulated in objects
- **Information** hiding
  - used to protect internal properties of an object
- Objects interact by means of **message passing**
  - a metaphor for applying an operation on an object
    - ...but it was not meant to be a metaphor at the beginning...
- In object-oriented languages objects are grouped in **classes**
  - classes represent concepts whereas objects represent phenomena
  - object-based or prototype based languages => no classes
    - e.g. JavaScript, Self
- **Inheritance**
  - classes are organized in inheritance hierarchies
  - provides for class *extension* or *specialization*

MULTI-PARADIGM APPROACHES

- Problem/Motivation
  - no one paradigm solves all problems in the easiest or most efficient way
- Idea
  - *more programming paradigms in the same language*
  - providing a framework in which programmers can work in a variety of styles
    - freely intermixing constructs from different paradigms
    - allowing programmers to use the best tool for a job
- Problems
  - integrating different models of computation and programming models

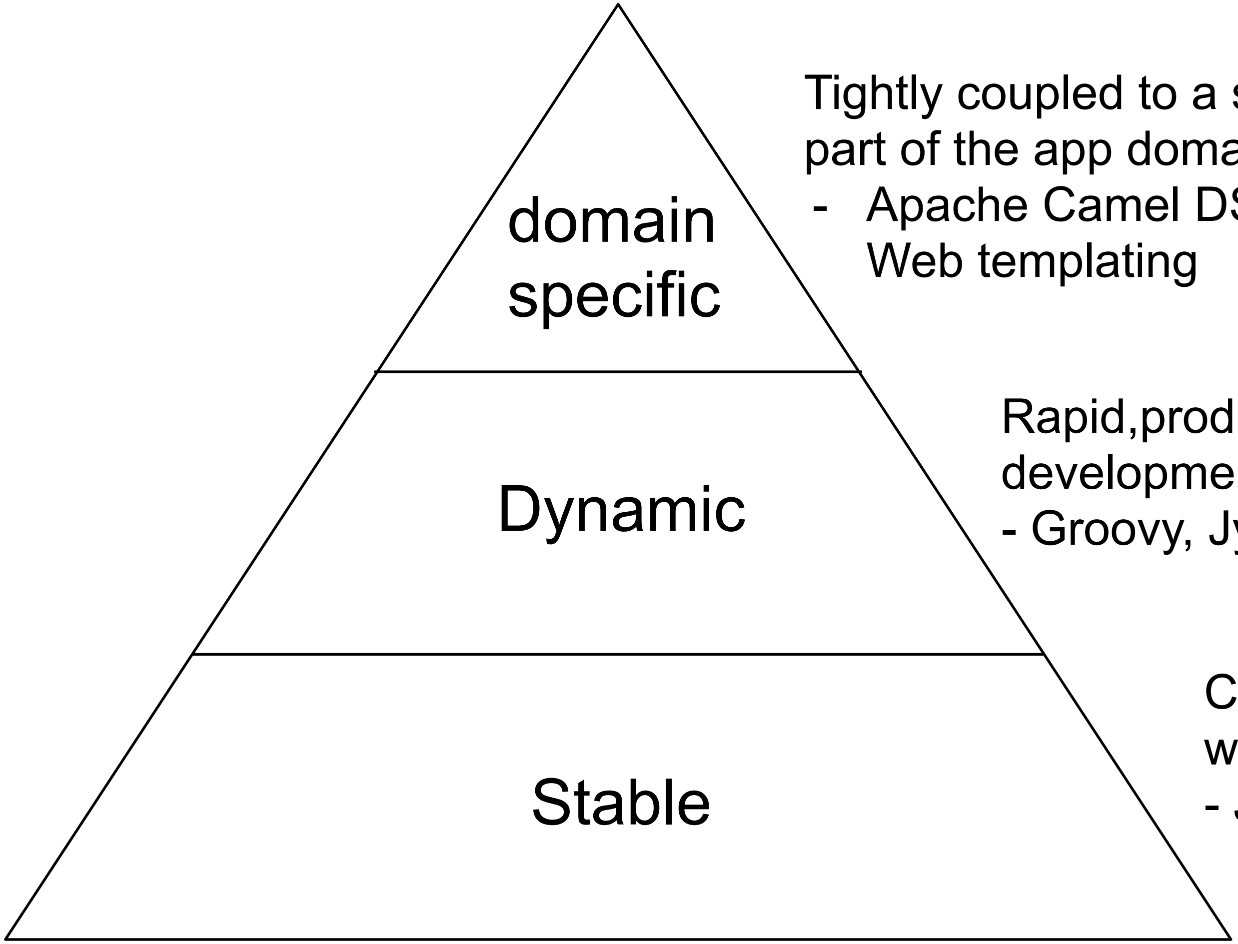
MULTI-PARADIGM APPROACHES

- Examples
  - **OOP + Functional**
    - JavaScript, Python, C#, Java 8, …
    - Scala
  - Oz
    - logic + functional + data-flow concurrent
  - Alice, Curry, CIAO

POLYGLOT VIRTUAL MACHINES

- .NET CLR
  - explicitly designed from scratch to support multiple languages of different paradigms
  - main languages: C#, VisualBasic, F#,
- JVM
  - originally designed for a single OOP language
  - however many JVM-based languages developed on top
    - Scala, Groovy, Clojure, JRuby, Jython, ...
  - recent language extension to integrate functional programming
    - project Lambda - Java 8
    - but without changing the JVM specification

POLYGLOT PROGRAMMER PYRAMID



domain specific

Tightly coupled to a specific part of the app domain

- Apache Camel DSL, Drools, Web templating

Dynamic

Rapid,productive, flexible development of functionality
- Groovy, Jython, Clojure

Stable

Core functionality, stable, well tested, performant
- Java, Scala

(From “Well-Grounded Java Developer” - Evans, Verburg - Ch. 7 - Alternative JVM languages)

POLYGLOT PROGRAMMER PYRAMID

Name	Example problem domain
Domain Specific	Build, continuous integration, continuous deployment Dev-ops Enterprise Integration Pattern modeling Business Rule Modelling
Dynamic	Rapid Web Development Prototyping Interactive administrative and user consoles Scripting Tests
Stable	Concurrent code Application containers Core business functionalities

MULTI-PARADIGM APPROACHES

- A further approach: *coordination models and languages* [Gelernter & Carriero]
  - given a system as an ensemble of interacting entities, then:
    - each entity maybe designed and developed according to some specific paradigm
    - common language used to express and enable interaction and coordination among entities
      - e.g. Tuple Space model & Linda language
  - based on the *orthogonality* between computation and interaction/coordination

THE RISE OF CONCURRENT AND ASYNCHRONOUS PROGRAMMING

THE RISE OF CONCURRENCY

- What about **concurrent programming**? including...
  ... parallel programming
  ... asynchronous/event-driven programming
  ... distributed programming
  ... real-time/time-oriented programming
- Is it concurrent programming a paradigm? Are these paradigms?
  - can be conceived just as extensions of existing paradigms?

TERMINOLOGY

- **Concurrent** programming
  - building programs in which multiple computational activities *overlap* in time and typically interact in some way
    - without necessarily running on separate physical processors
  - logical/abstract/programming level
- **Parallel** programming
  - the execution of programs overlaps in time by running on *separate physical processors*
  - physical level
- **Distributed** programming
  - when processors are distributed over a network
  - no shared memory

CONCURRENCY “PARADIGMS”

- **Multi-threaded programming**
  - shared state
  - synchronization mechanisms
    - semaphores, monitors
- **Message-based programming**
  - no shared state
  - interaction by means of message exchange
- **Event-driven programming**
  - the flow of the program is determined by events
    - user actions (mouse clicks, key presses), sensors, messages from other threads/process/apps

CONCURRENCY “PARADIGMS”

- **Asynchronous programming**
  - designing programs featuring asynchronous actions and requests
    - never blocking dogma
    - future mechanisms, callbacks
- **Reactive programming**
  - the flow of the program is designed around data flows and the propagation of change

IMPACT OF CONCURRENCY ON PARADIGMS

- Existing paradigms + concurrency mechanisms
  - multi-threaded programming
    - e.g. Java
- Integrating concurrency *within* the paradigm => new paradigm
  - example: OOP + concurrency

    => **actors** & concurrent objects

    => active objects

    => other flavors of concurrent OOP
    - SCOOP model in Eiffel
  - example: Functional + actors
    - Erlang

BEYOND TURING MACHINES

- New models of computation
  - process algebra
    - CSP, CCS, π-calculus
  - Petri-nets
  - chemical abstract machines
  - …
- Key point: ***interaction*** [Milner,Wegner]
  - which cannot be properly captured by pure computational model such as λ-calculus or Turing machines

LANGUAGES vs. FRAMEWORKS/ LIBRARIES

- Languages
  - first-class concurrent abstractions are first-class constructs of the language
    - Erlang
- Libraries/Frameworks
  - first-class concurrent abstractions are represented by existing abstractions of a host language
    - e.g. Java/Scala + Actor Library
  - frameworks define the general organization of a program and its lifecycle

STATE-OF-THE-ART & RESEARCH LANDSCAPE

- Active Objects and Actors
- Software Transactional Memory
- Reactive programming
- Agents
- ...

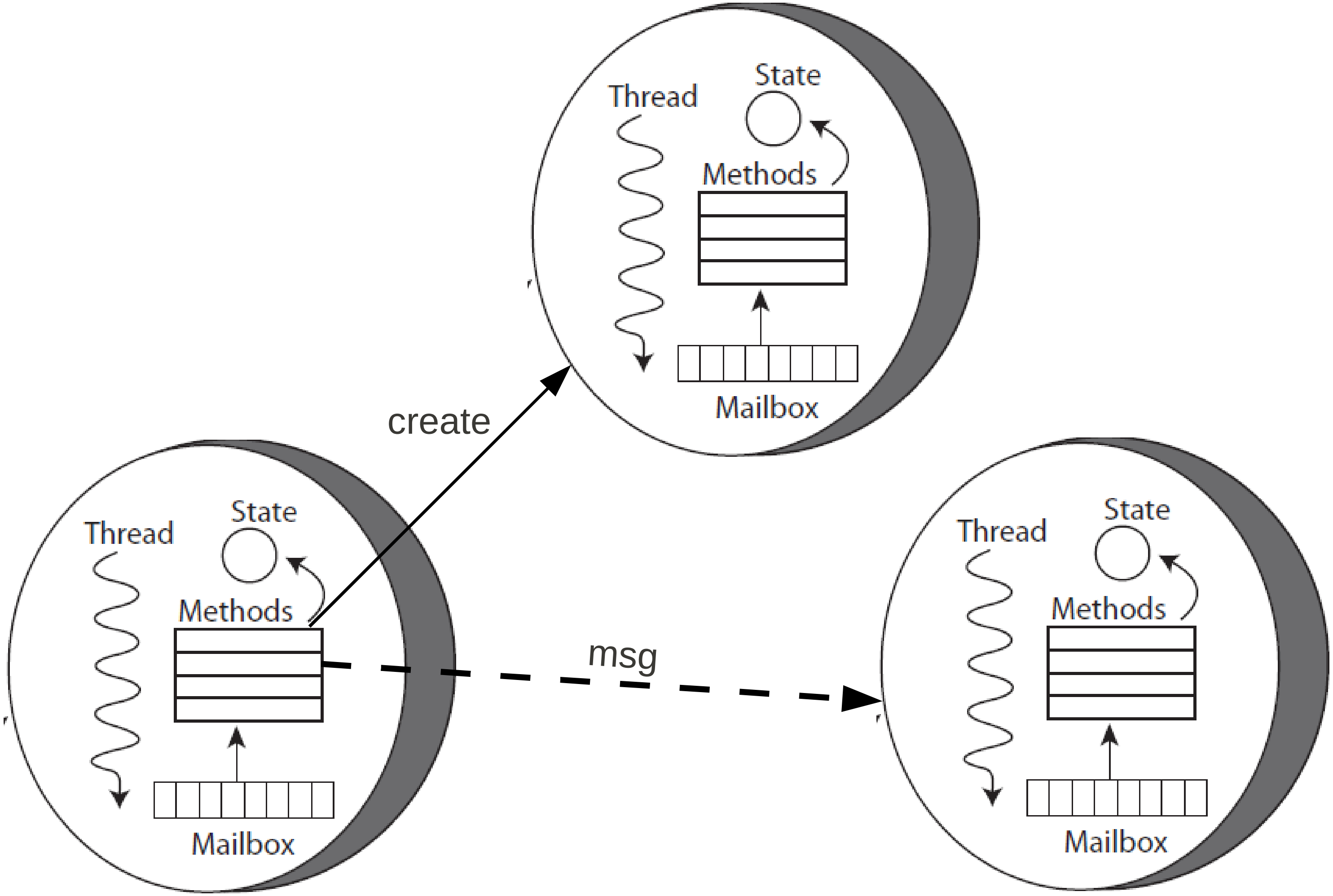
ACTOR MODEL

- Originally introduced by Carl Hewitt and colleagues at MIT in 70ies
  - AI context
- Developed by Gul Agha, Akinori Yonezawa et al. in 80ies and 90ies as the unification of OOP and concurrency
  - many languages & frameworks
    - ACT++, Salsa, Kilim, ABCL family, E, AmbientTalk, ActorFoundry,...
- Playing a major role in the mainstream nowadays
  - as an alternative model to multi-threaded programming
  - Erlang, Scala/Akka actors, HTML5 Web Workers, DART isolates, etc.

ACTOR MODEL

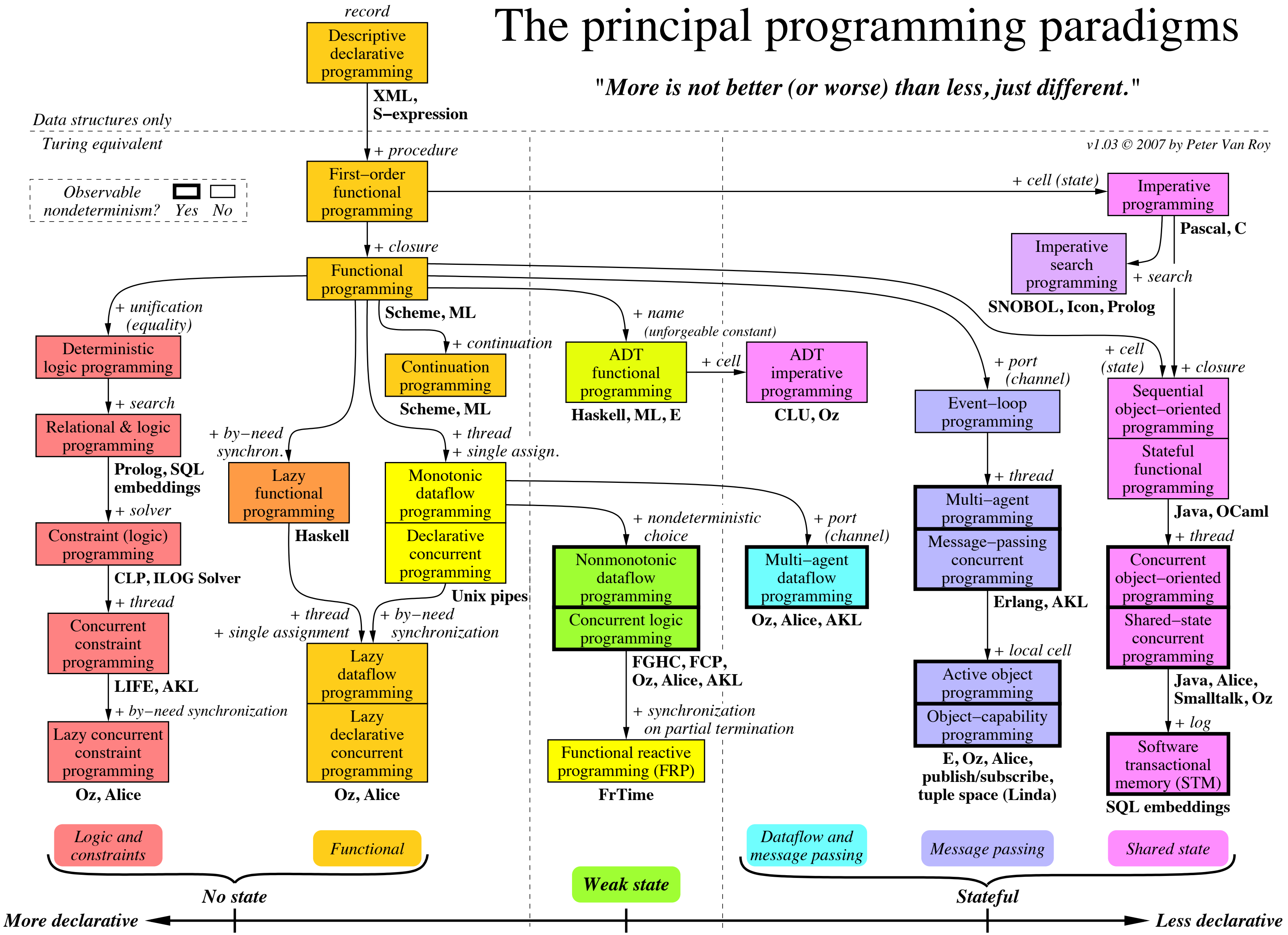
- *Asynchronous* message passing among autonomous *purely reactive* objects called *actors*
  - everything is an actor
    - with a unique identifier
    - a unique mailbox where messages are enqueued
  - every interaction takes place as async message passing
- Few primitives
  - **send**, **create**, **become**
- *Everything - including traditional control structures, can be modeled as patterns of messages among actors*

ACTOR MODEL


Thread
State
Methods
Mailbox
create
msg
Thread
State
Methods
Mailbox
Thread
State
Methods
Mailbox

UNDERSTANDING PARADIGM RELATIONSHIPS => BUILDING A TAXONOMY

VAN ROY'S TAXONOMY



The principal programming paradigms

"More is not better (or worse) than less, just different."

v1.03 © 2007 by Peter Van Roy

Data structures only

Turing equivalent

Observable nondeterminism? ▭ *Yes* ▭ *No*

record

Descriptive declarative programming

XML, S−expression

+ procedure

First−order functional programming

+ cell (state)

Imperative programming

Pascal, C

+ search

Imperative search programming

SNOBOL, Icon, Prolog

+ closure

Functional programming

Scheme, ML

+ 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

+ continuation

Continuation programming

Scheme, ML

+ by−need synchron.

Lazy functional programming

Haskell

+ thread
+ single assign.

Monotonic dataflow programming

Declarative concurrent programming

Unix pipes

+ thread
+ single assignment

+ by−need synchronization

Lazy dataflow programming

Lazy declarative concurrent programming

Oz, Alice

+ name (unforgeable constant)

ADT functional programming

Haskell, ML, E

+ cell

ADT imperative programming

CLU, Oz

+ nondeterministic choice

Nonmonotonic dataflow programming

Concurrent logic programming

FGHC, FCP, Oz, Alice, AKL

+ synchronization on partial termination

Functional reactive programming (FRP)

FrTime

+ port (channel)

Multi−agent dataflow programming

Oz, Alice, AKL

+ port (channel)

Event−loop programming

+ thread

Multi−agent programming

Message−passing concurrent programming

Erlang, AKL

+ local cell

Active object programming

Object−capability programming

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

Java, Alice, Smalltalk, Oz

+ log

Software transactional memory (STM)

SQL embeddings

Logic and constraints · *Functional* · *Weak state* · *Dataflow and message passing* · *Message passing* · *Shared state*

No state · *Stateful*

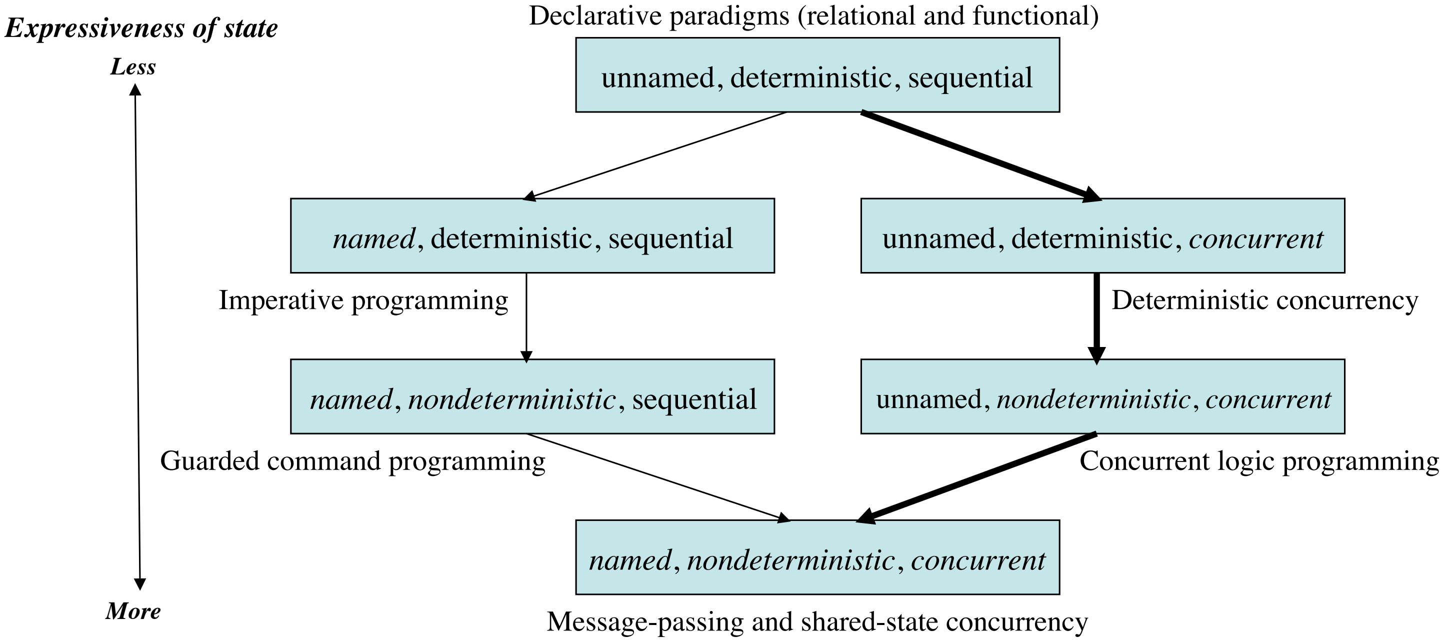
More declarative ← → *Less declarative*

OBSERVABLE NONDETERMINISM

- The first key property of a paradigm is whether or not it can express *observable nondeterminism.*
- **Non-determinism** = *when the execution of a program is not completely determined by its specification*
  - at some point during the execution the specification allows the program to choose what to do next.
- ***Observable* non-determinism** => when a user can see different results from executions that start at the same internal configuration
  - highly undesirable
    - a typical effect is a race condition = where the result of a program depends on precise differences in timing between different parts of a program (a “race”)
- Observable non-determinism should be supported only if its expressive power is needed.
  - especially true for concurrent programming.

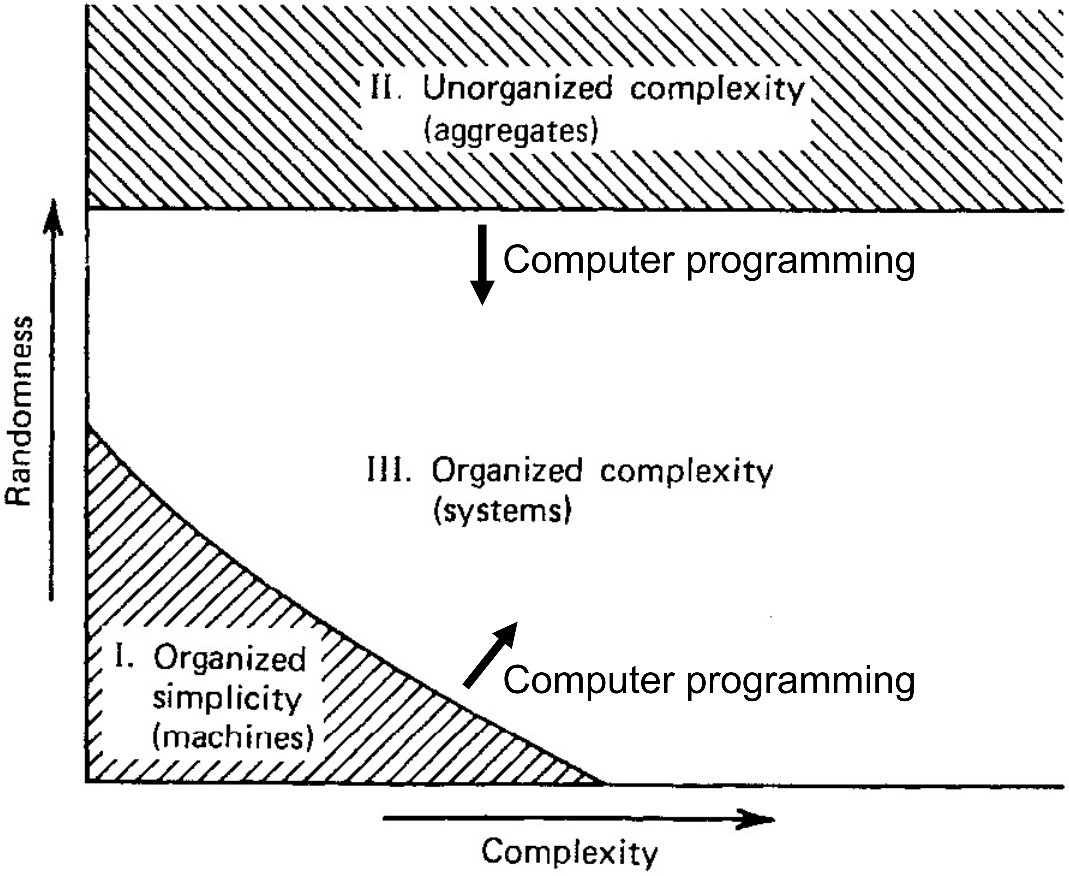
NAMED STATE

- The second key property of a paradigm is *how strongly it supports state*
- State is the ability to remember information, or more precisely, to store a sequence of values in time
  - its expressive power is strongly influenced by the paradigm that contains it



COMPUTER PROGRAMMING & SYSTEM DESIGN

- Van Roy’s diagram about the view of computer programming in the context of general system design
  - Weinberg’s diagram + computer programming



COMPUTER PROGRAMMING & SYSTEM DESIGN

- Axes => two main properties of systems:
  - complexity
    - the number of basic interacting components
  - randomness
    - how nondeterministic the system's behavior is
- There are two kinds of systems that are understood by science:
  - aggregates
    - e.g., gas molecules in a box, understood by statistical mechanics
  - machines
    - e.g., clocks and washing machines, a small number of components interacting in mostly deterministic fashion
- The large white area in the middle is mostly not understood

COMPUTER PROGRAMMING & SYSTEM DESIGN

- The *science* of computer programming is pushing inwards the two frontiers of system science
  - computer programs can act as highly complex machines and also as aggregates through simulation.
  - computer programming permits the construction of the most complex systems
- We would therefore like to understand them in a scientific way
  - by understanding the basic concepts that compose the underlying paradigms and how these concepts are designed and combined

WHEN A NEW PARADIGM IS NEEDED: CREATIVE EXTENSION PRINCIPLE

- Question
  - when a new paradigm is needed?
  - when a new feature of a language brings a new paradigm?
- ***Creative Extension Principle*** by Felleisen & Van Roy
  - in a given paradigm, it can happen that programs become complicated for *technical reasons that have no direct relationship to the specific problem that is being solved*
  - this is a sign that there is a new concept waiting to be discovered

CREATIVE EXTENSION PRINCIPLE - EXAMPLE

- Starting point
  - simple *sequential functional* programming paradigm
- three scenarios of how new concepts can be discovered and added to form new paradigms
  - state
  - concurrency
  - exception

SECOND SCENARIO: ADDING STATE

- Need
  - modeling updatable memory
    - entities that remember and update their past
- Solution
  - adding two arguments to all function calls relative to that entity
    - the arguments represent the input and output values of the memory
    - this is unwieldy and it is also not modular because the memory travels throughout the whole program
- New concept that wants to come out
  - *state*

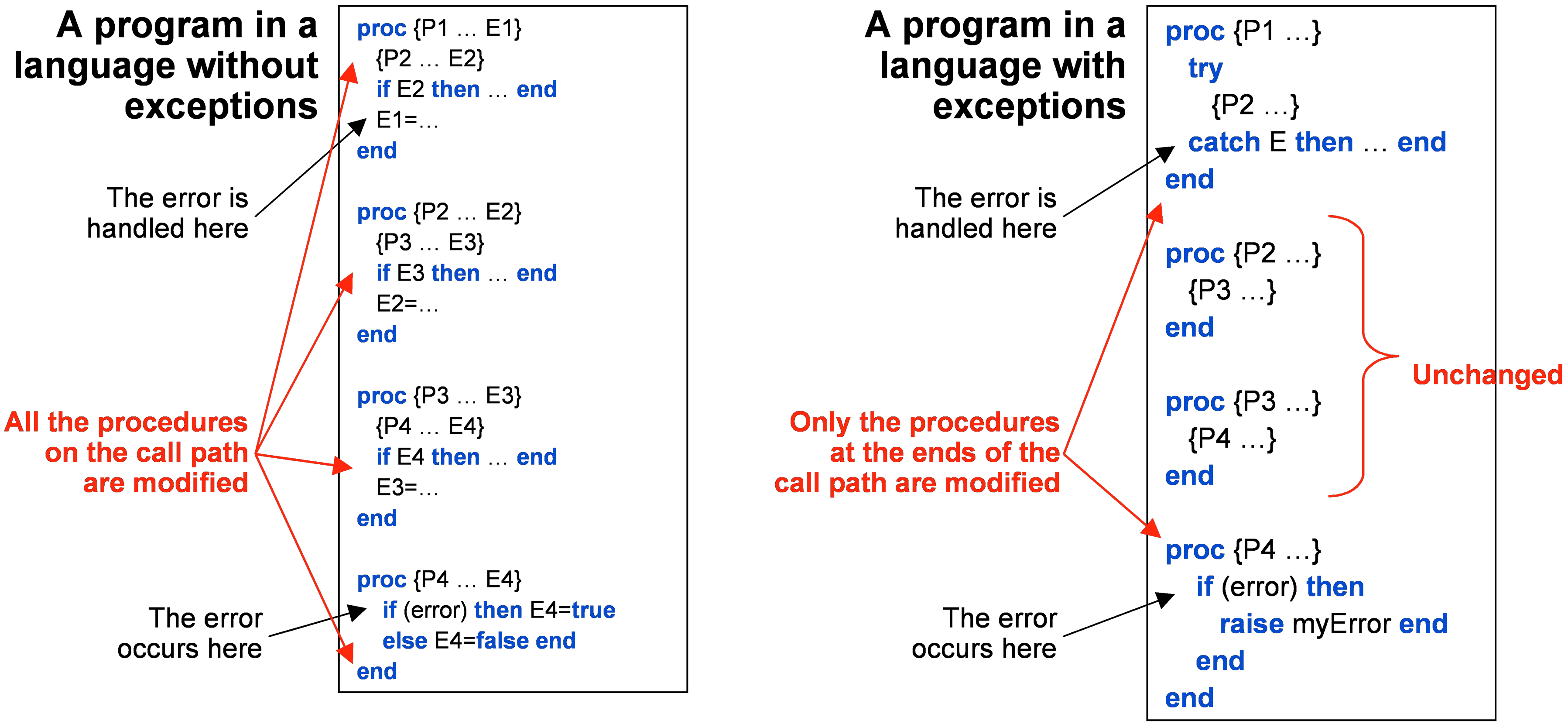
FIRST SCENARIO: ADDING CONCURRENCY

- Need
  - modeling several independent activities
- Solution
  - adding several execution stacks, a scheduler, and a mechanism for preempting execution from one activity to another
- New concept that wants to come out
  - *concurrency*

THIRD SCENARIO: ADDING EXCEPTIONS

- Need
  - modeling error detection and correction
  - any function can detect an error at any time and transfer control to an error correction routine
- Solution
  - adding error codes to all function outputs and conditionals to test all function calls for returned error codes
- New concept that wants to come out
  - *exceptions*

THIRD SCENARIO: ADDING EXCEPTIONS



A program in a language without exceptions

```
proc {P1 … E1}
  {P2 … E2}
  if E2 then … end
  E1=…
end

proc {P2 … E2}
  {P3 … E3}
  if E3 then … end
  E2=…
end

proc {P3 … E3}
  {P4 … E4}
  if E4 then … end
  E3=…
end

proc {P4 … E4}
   if (error) then E4=true
   else E4=false end
end
```

The error is handled here

All the procedures on the call path are modified

The error occurs here

A program in a language with exceptions

```
proc {P1 …}
  try
    {P2 …}
  catch E then … end
end

proc {P2 …}
  {P3 …}
end

proc {P3 …}
  {P4 …}
end

proc {P4 …}
   if (error) then
     raise myError end
   end
end
```

The error is handled here

Only the procedures at the ends of the call path are modified

Unchanged

The error occurs here

DISCOVERING NEW PARADIGMS

- The common theme in these three scenarios is that *we need to do pervasive (nonlocal) modifications of the program in order to handle a new concept*
  - if the need for pervasive modifications manifests itself, we can take this as a sign that there is a new concept waiting to be discovered
- By adding this concept to the language we no longer need these pervasive modifications and we recover the simplicity of the program.
  - the only complexity in the program is that needed to solve the problem
  - no additional complexity is needed to overcome technical inadequacies of the language.

BIBLIOGRAPHY

- Van Roy, Haridi. “Concepts, Techniques, Models of Computer Programming”, MIT Press.
- Van Roy. “Programming Paradigms for Dummies: What Every Programmer Should Know” In G. Assayag and A. Gerzso (eds.) New Computational Paradigms for Computer Music, IRCAM/Delatour, France.
- “Overview of the main programming paradigms”, Kurt Nørmark, Department of Computer Science, Aalborg University, Denmark (http://people.cs.aau.dk/~normark/prog3-03/html/notes/theme-index.html)
- David Gelernter and Nicholas Carriero. 1992. Coordination languages and their significance. Commun. ACM 35, 2 (February 1992), 97-107.
- Herb Sutter and James Larus. 2005. Software and the Concurrency Revolution. Queue 3, 7 (September 2005), 54-62
- Robin Milner. 1993. Elements of interaction: Turing award lecture. Commun. ACM 36, 1 (January 1993), 78-89
- Peter Wegner. 1997. Why interaction is more powerful than algorithms. Commun. ACM 40, 5 (May 1997), 80-91
- Felleisen M., “On the Expressive Power of Programming Languages", in 3rd European Symposium on Programming (ESOP 1990), May 1990, pp. 134-151