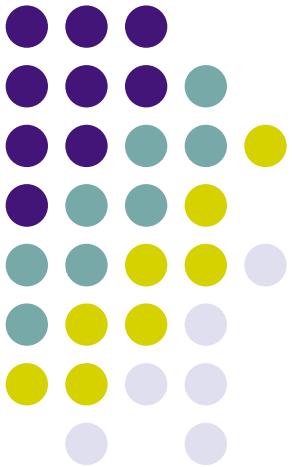
Introduction to programming paradigms

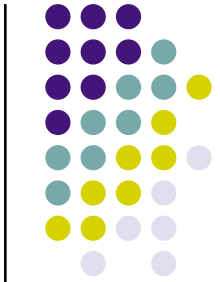
August 15, 2013

Universidad del Valle
AVISPA Research Group

Peter Van Roy
Université catholique de Louvain
Louvain-la-Neuve, Belgium

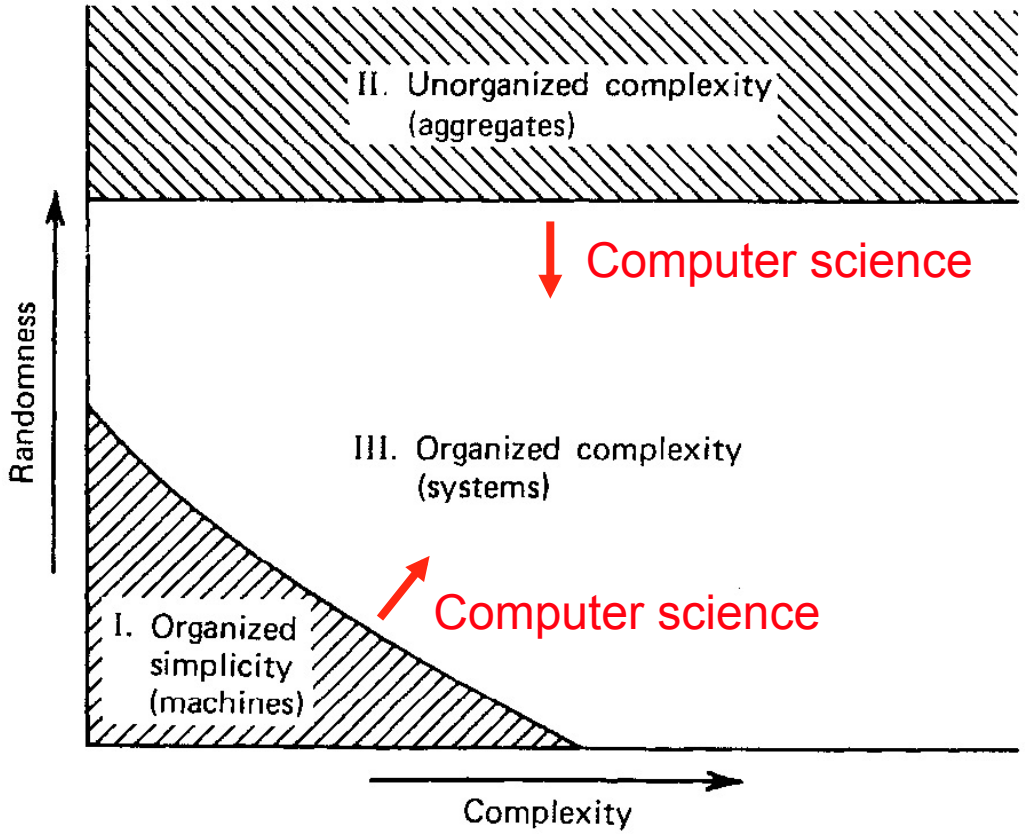


Programming paradigms



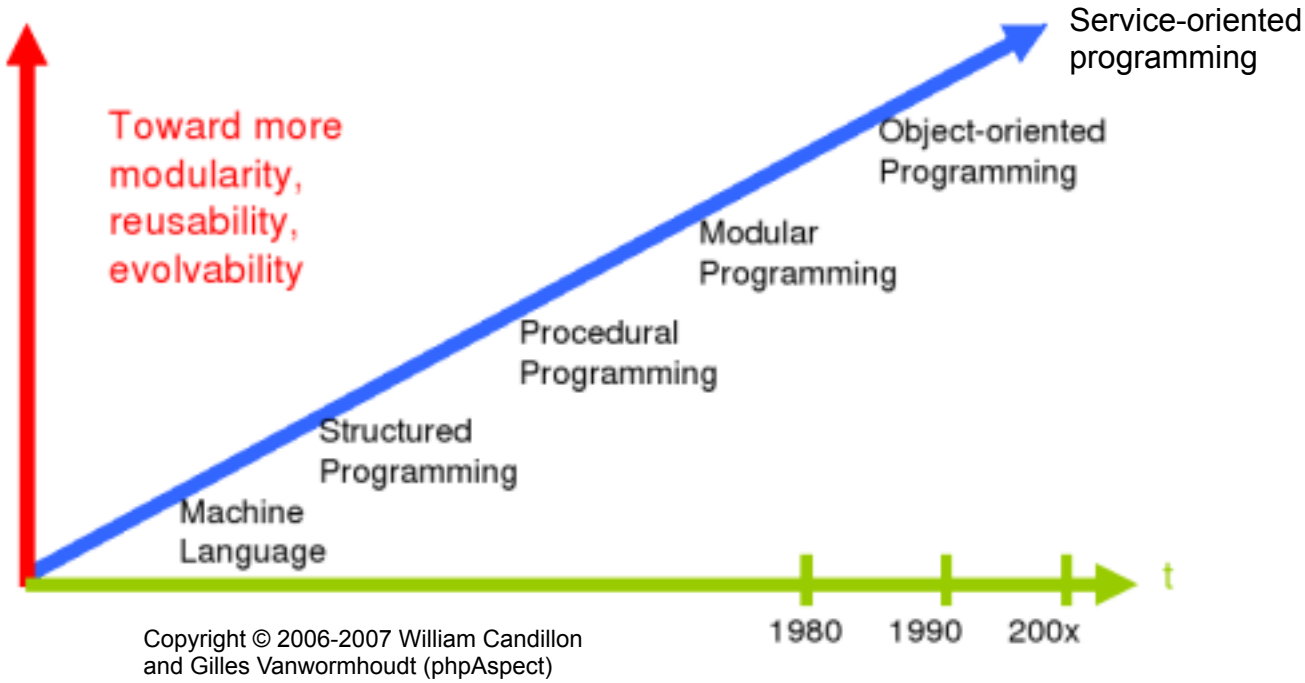
- A paradigm is *an approach to programming a computer based on a coherent set of principles or a mathematical theory*
  - Different theories of computing result in different paradigms (λ calculus, π calculus, first-order logic, Hoare logic, …)
  - No existing theory covers all programming concepts!
  - Programming is truly a new discipline that is not covered by traditional mathematical theories
- We will introduce the world of programming paradigms
  - Why do we need many paradigms?
  - Because solving a problem is much easier when done in the right paradigm!

Programming and complex systems



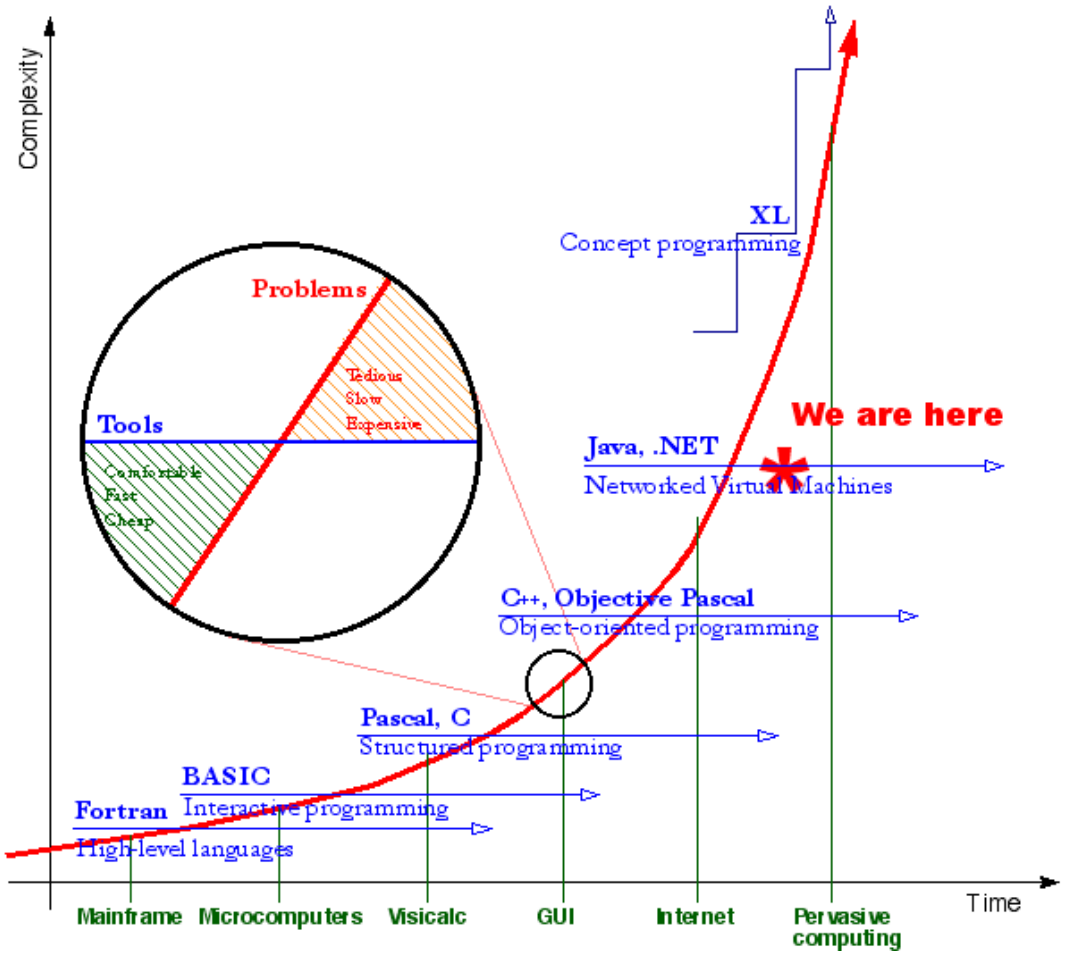
- Systems are composed of parts that interact in a well-defined way to provide a new behavior
  - This diagram comes from [Weinberg 1977] *An Introduction to General Systems Thinking*
- Computer science is the most advanced discipline for building complex systems
- Programming paradigms are the vanguard of building complex systems
- Each newly discovered programming concept advances the science of building complex systems

Traditional view of programming progress (1)



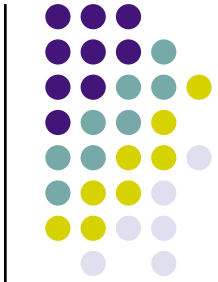
- This is a typical diagram illustrating how programming has progressed
- This diagram only says a little bit; it leaves out many important ideas

Traditional view of programming progress (2)



- This diagram is more realistic
- Each new paradigm simplifies programming: it raises the level of complexity that is easy to program
- Again, this diagram does not take into account many important programming concepts
  - At least, distributed (networked) programming is mentioned, which is part of the paradigm of concurrent programming!

The paradigm jungle



Deterministic logic programming

Transactional programming

First-order functional programming

Active object programming

Lazy functional programming

Functional reactive programming

Functional programming

Imperative programming

Relational programming

Descriptive declarative programming

Concurrent constraint programming

Constraint programming

Multiagent programming

Multiagent dataflow programming

Event-loop programming

Monotonic dataflow programming

Sequential object-oriented programming

Concurrent object-oriented programming

Lazy dataflow programming

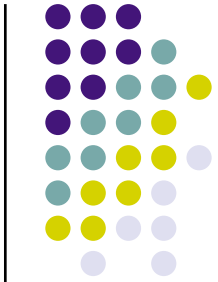
Nonmonotonic dataflow programming

Lazy constraint programming

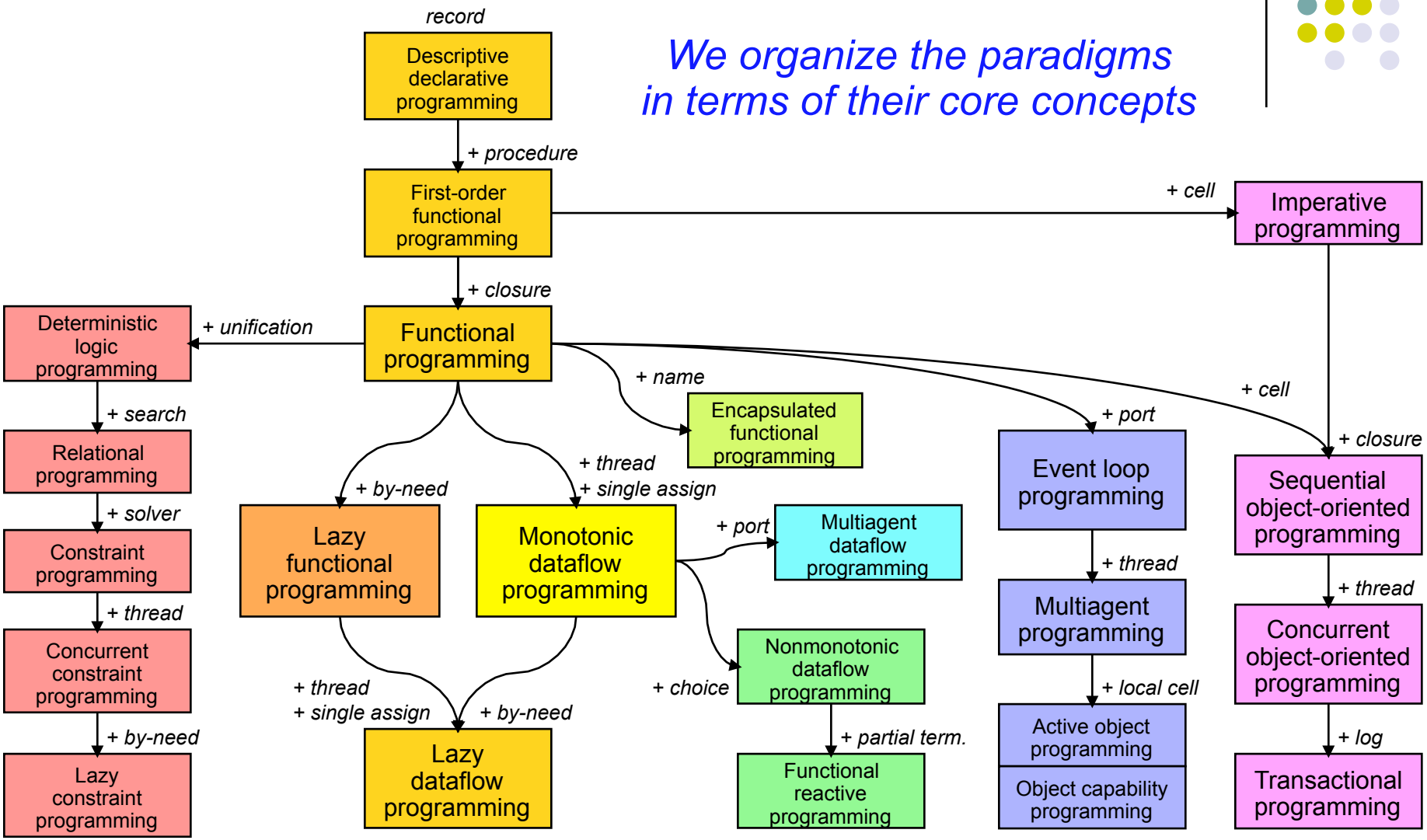
Encapsulated functional programming

Object capability programming

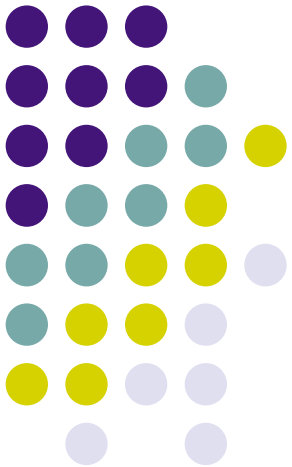
Taxonomy of paradigms



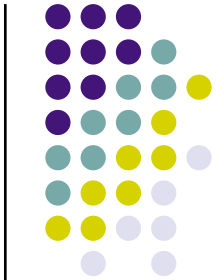
We organize the paradigms in terms of their core concepts



Concurrency and nondeterminism

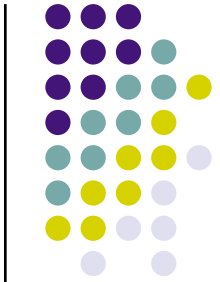


Concurrency



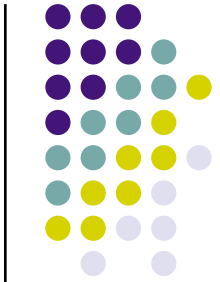
- The real world is concurrent
  - It consists of activities that evolve independently
- The computing world is concurrent too, at three levels
  - Distributed system: computers linked by a network
    - A concurrent activity is called a **computer**
  - Operating system of one computer
    - A concurrent activity is called a **process**
    - Each process has an independent memory space (competitive)
  - Activities inside one process
    - A concurrent activity is called a **thread**
    - All threads share the same memory space (cooperative)

Concurrency is really hard?



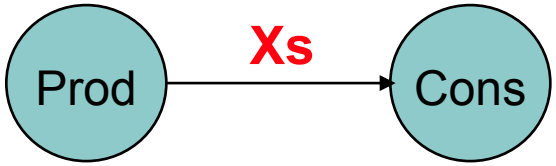
- At least, that's what people believe!
- It's true in C++ and Java: synchronized objects (monitors) are really hard to program with
- But the problem is not concurrency, it's monitors!
  - Monitors are the wrong paradigm for concurrency
- So what's the right paradigm?
  - That's what we'll see now!

The right paradigm: deterministic dataflow



- Producer/consumer pipeline

```
fun {Prod N Max}
   if N<Max then
      N|{Prod N+1 Max}
   else nil end
end
```

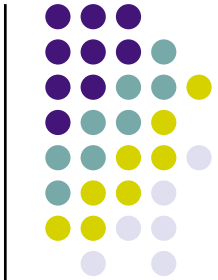



```
proc {Cons Xs}
   case Xs of X|Xr then
      {Display X}
      {Cons Xr}
   [] nil then skip end
end
```

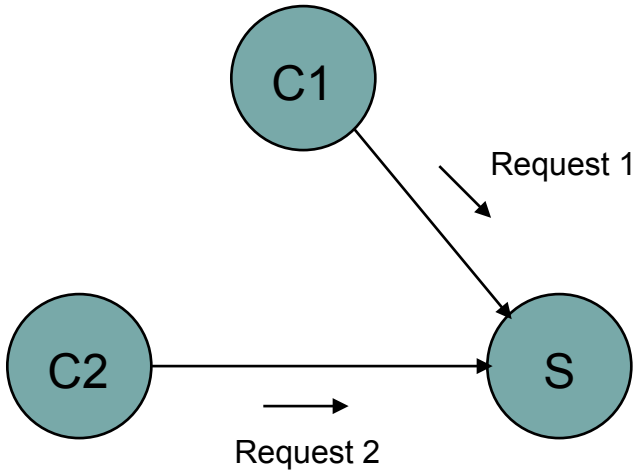
```
local Xs in
   thread Xs={Prod 0 1000} end
   thread {Cons Xs} end
end
```

- The threads in Prod and Cons share the dataflow variable **Xs**
- Dataflow behavior of the **case** (wait until data arrives) results in *stream communication*
- No other synchronization is needed!

Is deterministic dataflow enough?



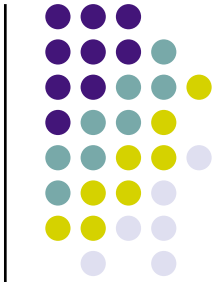
- Unfortunately not, sometimes it's not expressive enough
- Simple example: two clients talking to one server



C1 and C2 are independent: requests arrive at S in any order. If two requests arrive at the same time, S has to choose one of them.

This cannot be programmed with deterministic dataflow! S cannot receive messages in a fixed order because S does not know which client will send a message next. S must handle the messages as they arrive.

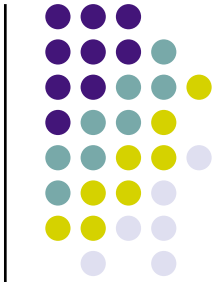
Nondeterministic choice



- Nondeterministic choice appears when two or more entities (clients) interact with the same entity (server)
  - Each client interacts with the server independently of the other clients
- The server must decide which message to handle first
  - The server (the runtime system, not the application) makes a choice
  - This choice is called nondeterminism

- Nondeterministic choice is a new programming concept
- Adding it gives a new paradigm, multiagent dataflow programming (which is similar to the actor model or message passing)

A few small examples…

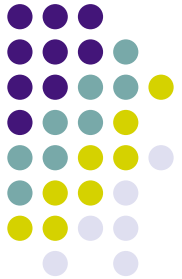


- Deterministic dataflow
  - Digital logic simulation (clocked execution)
  - Josephus problem (ring protocol)
  - Hamming problem (lazy protocol with cycle)
  - Bounded buffer (combining lazy and eager)

+ nondet. choice

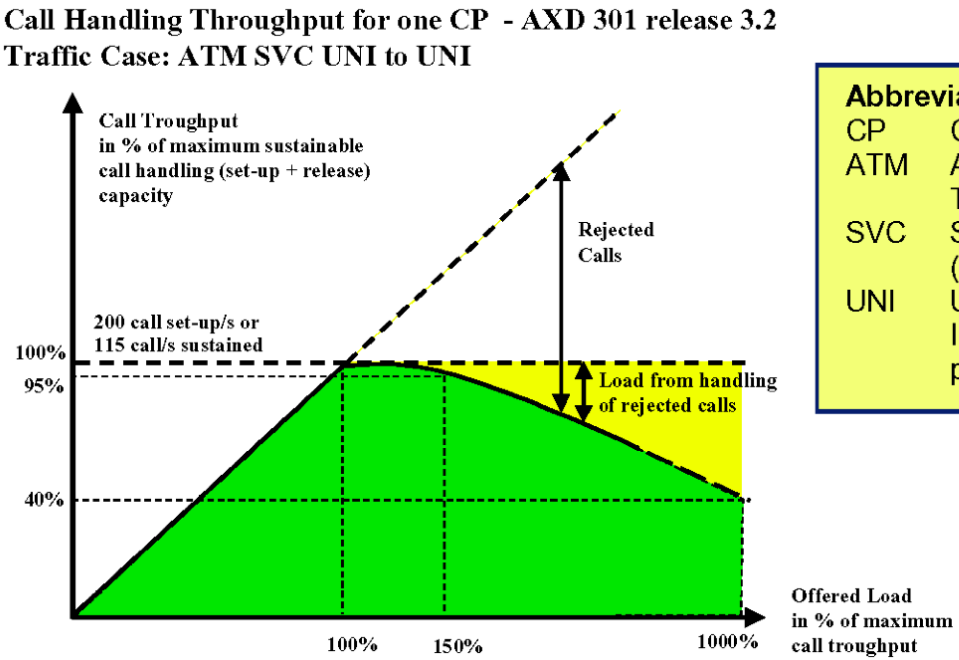
- Multiagent dataflow
  - Multiagent systems (agents talking to each other)
  - Distributed protocols (RMI and distributed algorithms)
  - Lift control system (real-world example multiagent system)

A bigger example…



- Ericsson AXD 301 ATM Switch: >1 million lines of Erlang
- **Erlang**: Concurrent and independent by default, asynchronous messages, multi-agent programs

- **Java**: Sequential and monolithic by default, synchronous RMI, shared-data programs

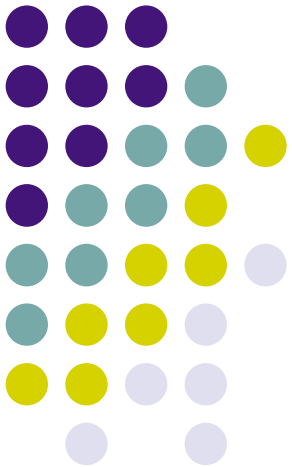


Abbreviations:

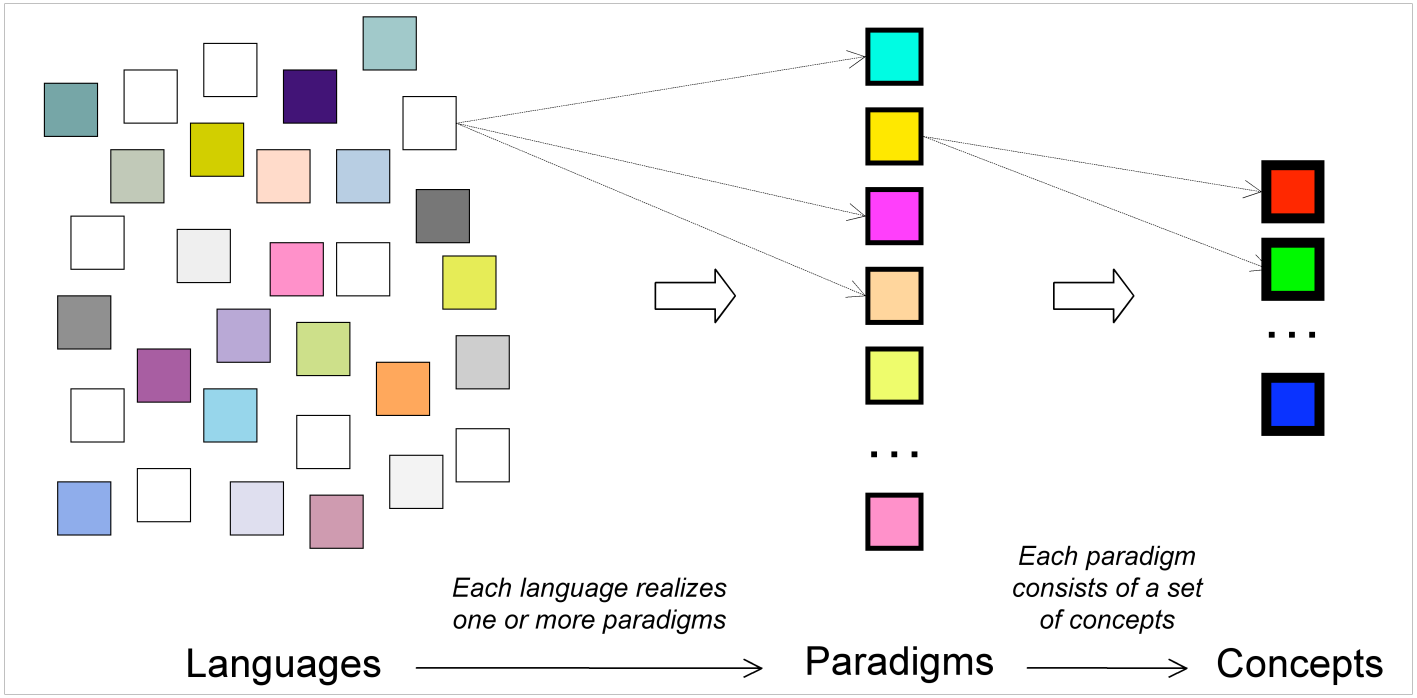
CP	Control Processor
ATM	Asynchronous Transfer Mode
SVC	Switched Virtual (ATM) Channel
UNI	User-Network Interface signaling protocol

- Object-oriented programming is the wrong paradigm for Internet programming!
  - Important: isolation, concurrency, asynchronous messages, higher-order programming
  - Unimportant: inheritance, classes, methods, UML diagrams, monitors

Paradigms and languages

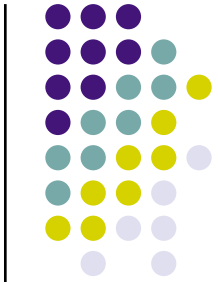


Paradigms, not languages



- There are lots of programming languages
  - Hundreds are used in industry: you will see very many in your careers!
- There are many fewer programming paradigms and concepts
  - Less than 30 paradigms are considered important
- If you want to understand programming, first understand paradigms!

The creative extension principle

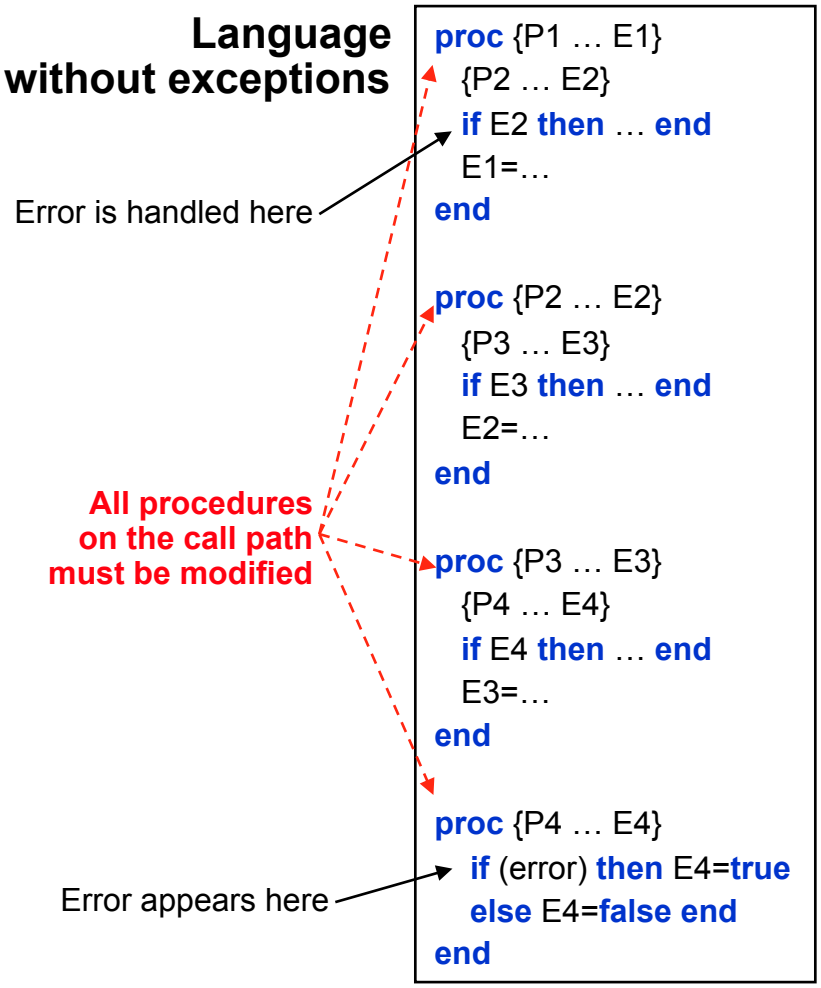


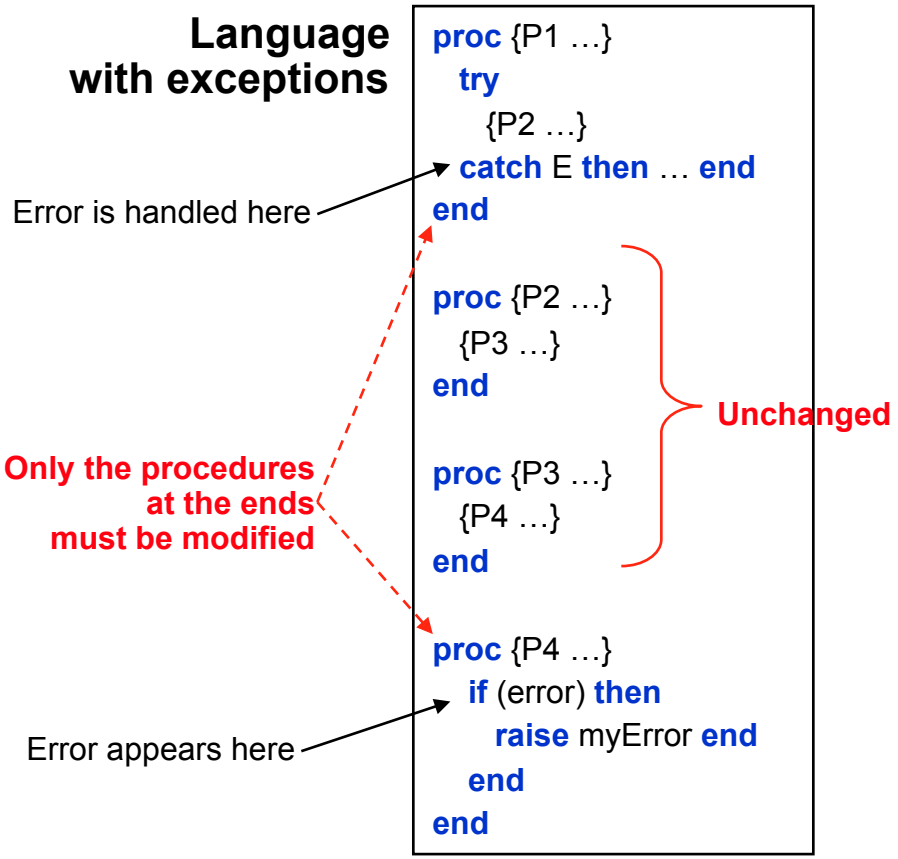
- How do we invent a new paradigm?

- When, in a given paradigm, programs start getting complicated for technical reasons that are unrelated to the problem being solved (e.g., nonlocal program transformations are needed), then there is a new programming concept waiting to be discovered!
  - Adding this concept to the paradigm lets programs get simple again

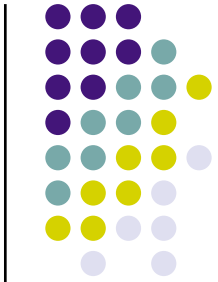
- We saw one example: the concept of nondeterministic choice
- Another example is the concept of exceptions
  - If the paradigm does not support them (e.g., like in C), then all routines on the call path must test and return error codes
  - If the paradigm supports them (e.g., like in Java), then only the ends of the call path need to be changed (raise and catch exceptions)

Exception handling





So which paradigm is best?



- Each is best for a particular kind of problem — the paradigm paradox
  - None is best overall: *"more is not better or worse, only different"*
- The conventional boundaries between paradigms are completely artificial (they exist only for historical reasons)
  - Java is only object-oriented ⇒ **wrong**
  - Scala is functional, object-oriented, and actor-based ⇒ **right**
- A big program almost always needs several paradigms
  - This is why you need to learn about multiple paradigms
- A good language should support several paradigms
  - This is hard for industry languages (Scala and Erlang are moving in the right direction; Java and C++ are stagnating)
  - In your course, you will use Oz: a research language that supports many paradigms (Oz ideas are slowly moving to industry…)