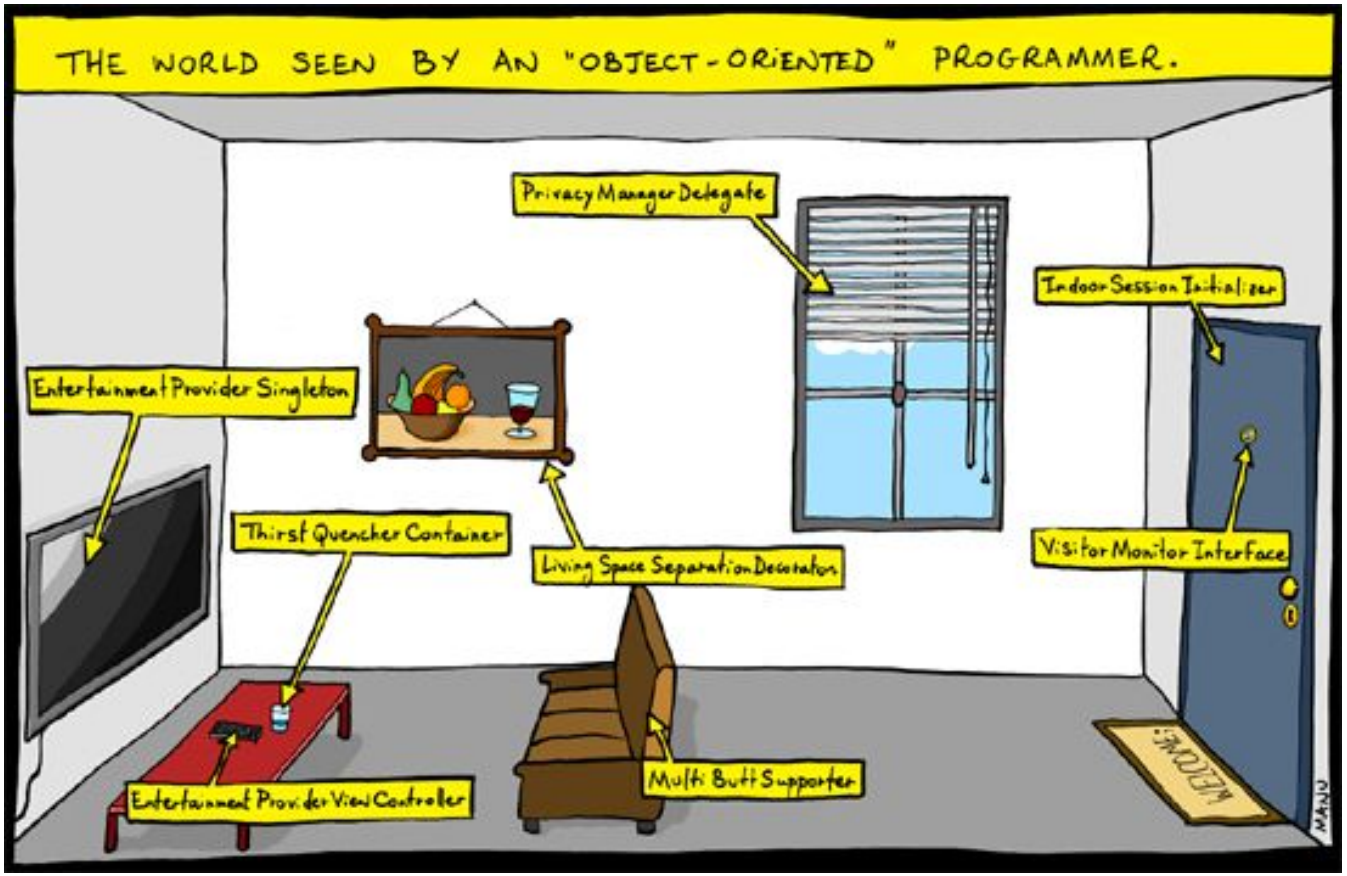
Lecture: Introduction & Administrivia

ESS201: Programming II

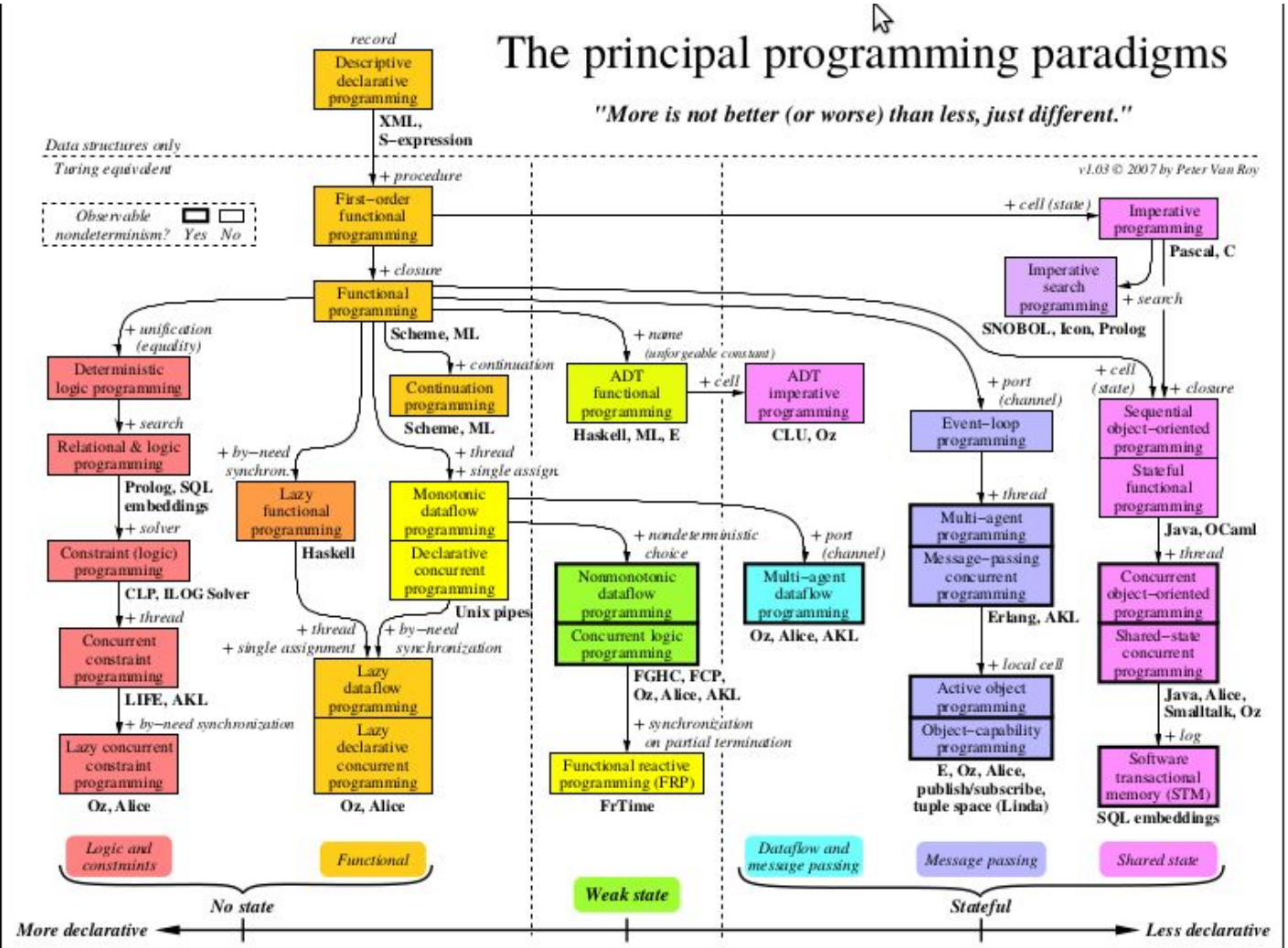
Jaya Sreevalsan Nair, IIIT-Bangalore

August 05, 2019

Introduction to Object-Oriented Programming



http://www.smashingapps.com/2011/09/18/the-world-seen-by-an-object-oriented-programmers-comic.html



Clarifications

This chart is inspired by "Concepts, Techniques, and Models of Computer Programming" (MIT Press, 2004).

The chart classifies programming paradigms according to their kernel languages (the small core language in which all the paradigm's abstractions can be defined). Kernel languages are ordered according to the creative extension principle: a new concept is added when it cannot be encoded with only local transformations. Two languages that implement the same paradigm can nevertheless have very different "flavors" for the programmer, because they make different choices on what programming techniques and styles to facilitate.

When a language is mentioned under a paradigm, it means that part of the language is intended (by its designers) to support the paradigm without interference from other paradigms. It does not mean that there is a perfect fit between the language and the paradigm. It is not enough that libraries have been written in the language to support the paradigm. The language's kernel language should support the paradigm. When there is a family of related languages, usually only one member of the family is mentioned to avoid clutter. The absence of a language does not imply any kind of value judgment.

Axes that are orthogonal to this chart are typing, aspects, and domain-specificity. Typing is not completely orthogonal: it has some effect on expressiveness. Aspects should be completely orthogonal, since they are part of a program's specification. A domain-specific language should be definable in any paradigm (except when the domain needs a particular concept).

Metaprogramming is another way to increase the expressiveness of a language. The term covers many different approaches, from higher-order programming, syntactic extensibility (e.g., macros), to higher-order programming combined with syntactic support (e.g., meta-object protocols and generics), to full-fledged tinkering with the kernel language (introspection and reflection). Syntactic extensibility and kernel language tinkering in particular are orthogonal to this chart. Some languages, such as Scheme, are flexible enough to implement many paradigms in almost native fashion. This flexibility is not shown in the chart.

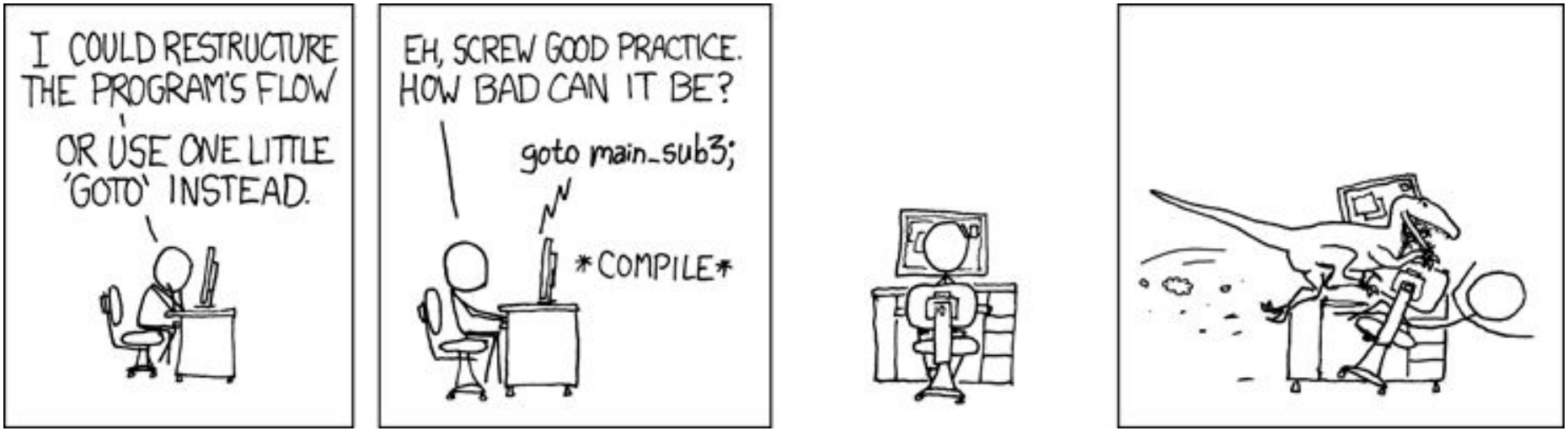
https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng101.pdf

Programming Paradigms

- Programming paradigm is a “way”/style of programming.
- **Imperative** Programming: Programming with an explicit sequence of commands that update state.
- **Declarative** Programming: Programming with a specific outcome, without a focus on “how” to do it.
- **Structured** Programming: Programming with clean, `goto`-free, nested control structures.
- **Procedural** Programming: Imperative programming with procedure calls.
- **Functional (Applicative)** Programming: Programming with functional call that avoid any global state.
- **Object-oriented** Programming: Programming by defining objects that send messages to each other.
- **Event-driven** Programming: Programming with emitters and listeners of asynchronous actions.

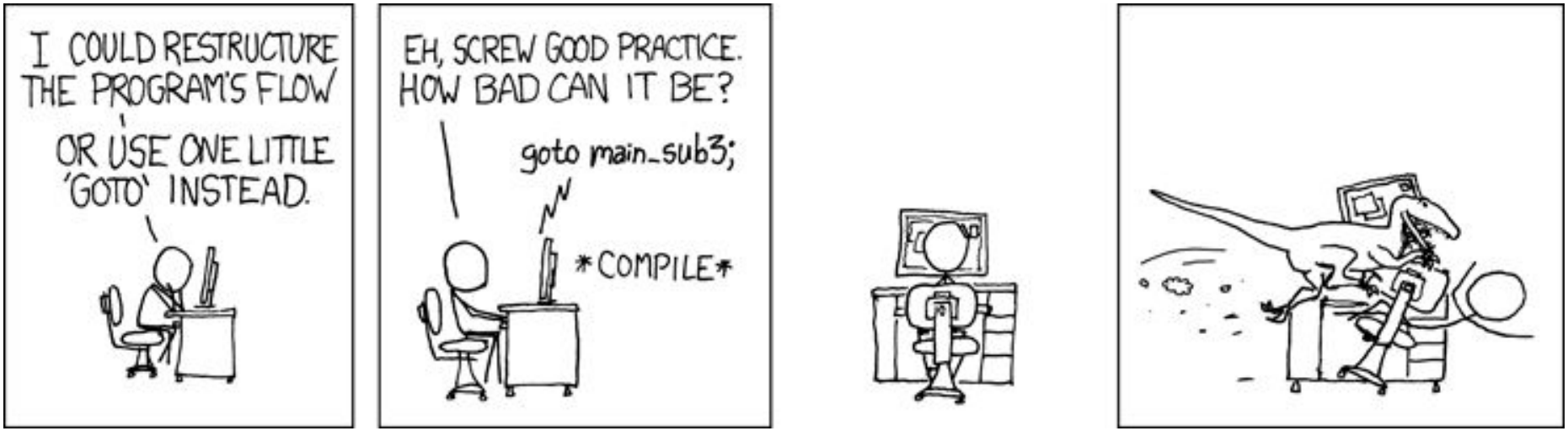
From https://cs.lmu.edu/~ray/notes/paradigms/

`goto` Considered Harmful?



https://xkcd.com/292/

`goto` Considered Harmful?



https://xkcd.com/292/

For more interested reading:

Edsger W. Djiksta (1968), "Letter to the editor: go to considered harmful", CACM, 11(3), pp 147-148, March 1968. (ACM DL url)

Donald E. Knuth (1974), "Structured Programming with go to statements", Journal ACM Computing Surveys (CSUR), 6(4), pp 261-301, December 1974. (ACM DL url)

Example

- A 3-dimensional point using struct, and building data structures further on it.

```
typedef struct {
     float x;
     float y;
     } point;
```

```
typedef struct {
     point ver1;
     point ver2;
     point ver3;
     } triangle;
```

Example

- A 2-dimensional point using struct, and building data structures further on it.

```
typedef struct {
    float x;
    float y;
    } point;
```

```
typedef struct {
    point ver1;
    point ver2;
    point ver3;
    } triangle;
```

- What's missing here? How would we write a function for a line-triangle intersection?

Example

- A 2-dimensional point using struct, and building data structures further on it.
- What's missing here? How would we write a function for a line-triangle intersection?
  - Introduce Object-oriented programming as a solution!
  - Classes and objects -- encapsulation, inheritance

Course Logistics

Course Objectives

- Improving problem solving and programming skills in Java and C++
- Understanding of object-oriented design
- Proficiency in Java and C++

Course Instruction Administrivia

- Instructor (C++): Jaya Sreevalsan Nair jnair@iiitb.ac.in
- Instructor (Java): T. K. Srikanth tk.srikanth@iiitb.ac.in
- Teaching Assistants:
  - Ayush Saraswat ayush.saraswat@iiitb.org
  - Nihal Kudligi nihal.kudligi@iiitb.org
  - Nimisha Garg nimisha.garg@iiitb.org
  - Raghavan G. V. raghavan.gv@iiitb.org
  - Srinivasan Vijayaraghavan srinivasan.vijayraghavan@iiitb.org
  - Vivek Yadav vivek.lpc@gmail.com
- Class Timings: MF 9:15a-10:45p
- Lab Timing: W 1:30p-3:30p (will be confirmed before the class on Aug 6)
- Class/Lab Venue: R203

Course Assessment Administrivia

- For each part (Java and C++) --
  - Assignments: 10% of final grade -- 2.5 each for the best 4 of the weekly assignments
  - Tests (2 in-class programming tests): 10% of final grade
  - Final assignment/mini-project: 5%
  - Exam (midterm, finals -- one of these in each part): 25%

Course Assessment Administrivia

All programming submissions (assignments/tests/exam) will be evaluated in two parts:

1. Code submitted to online tool DomJudge. Should compile and run on the sample inputs at least. Partial marks for partial correctness. No marks for code that does not compile and/or is largely incomplete.
2. Evaluation of design and code quality by instructor/TAs.

Course Logistics

- Attendance:
  - It is mandatory for all classes and labs.
  - Attendance less than 80% will result in ineligibility to appear for exams.
- Lab assignments:
  - Start working in the lab, the due date is 1 week from publishing the assignment
  - All assignments must be submitted for the assessment. If any of the assignments are not submitted, there will be appropriate penalty.
  - There will be 5-6 assignments per part.
  - Bring charged laptops on the lab days.
  - Some labs/assignments will involve interim assessments at the end of the lab
- Exams:
  - Both mid-term and finals will have a combination of programming tests and questions based on lectures.

Anti-plagiarism Policy

There will be zero tolerance for any form of plagiarism or similar cheating for all deliverables in the course used for assessment.

All code submitted should be based on your own efforts.

Sharing code for others to use/copy will be treated as “abetting” cheating, and will also be penalized.

Penalty will be decided on a case-to-case basis, but will minimally result in a 0 for that assignment/test, and a further penalty equal to the grading weightage for that work. Repeat incidents could result in an F for the course.

We intend to use software tools to identify attempts at plagiarism.