Building Software

September 14, 2017

Byung-Gon Chun

(Slide credits: George Candea, EPFL)

Defining the Problem

- Answers key question #1:
  - *“What problem should the software solve ?”*
- Also known as
  - *product vision, vision statement, product definition, ...*
- Simple formula
  - *brief (no more than 2 pages)*
  - *no reference to possible solutions*
  - *stated in plain language, from the user’s point of view*

Formulating Requirements

- Answers key question #2:
  - *“What should the software do to solve the problem ?”*
- Also known as
  - *requirements doc, requirements definition, functional specification,…*
- Makes user’s requirements explicit
  - *acts as a contract between user and developer*
  - *avoids arguments with user and with other developers*
  - *error in functional spec → discard code and tests*

Good Functional Specifications

- All user tasks
  - *description, expected response time, success vs. failure*
- All system inputs
  - *source, accuracy, value range, frequency of arrival*
- All system outputs
  - *destination, accuracy, value range, format, etc.*
- All interfaces with the rest of the world
  - *hardware, software, communication interfaces*

Good Functional Specs

- Requirements are verifiable
  - *quantitative vs. qualitative assessment*
- Competing attributes can be resolved
  - *clear guidance on how to make tradeoffs*
- Clear connection to problem definition
  - *each item contributes to solving the problem*

Requirements Change

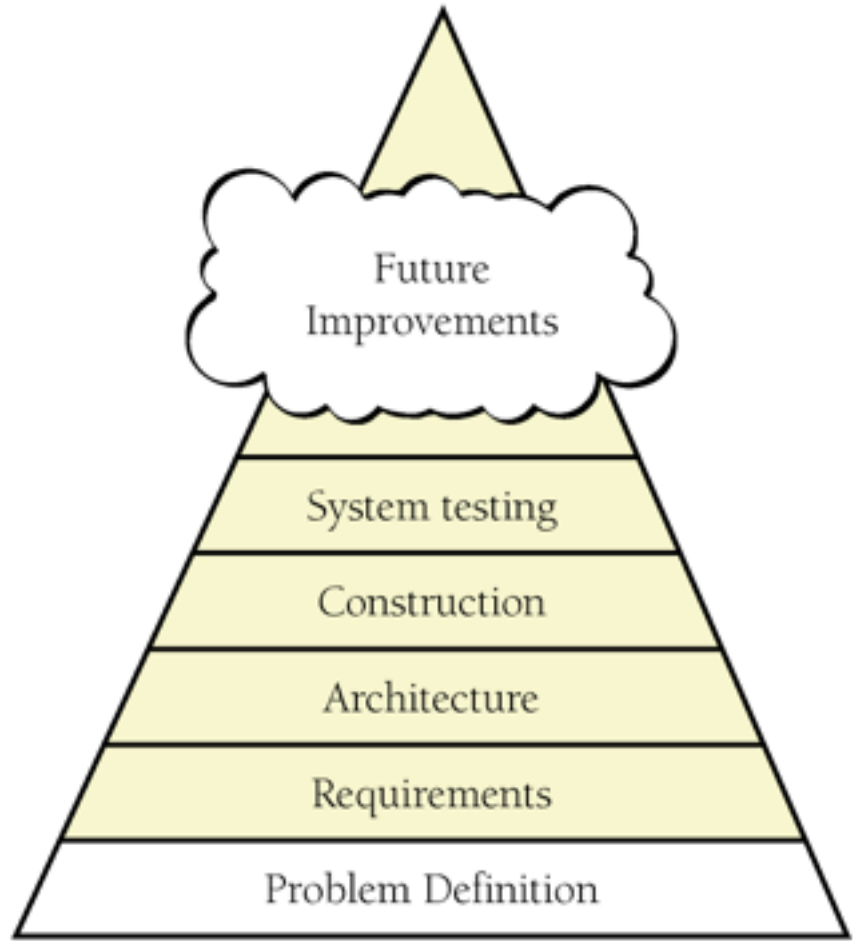
- Customer does not know what (s)he wants
  - *the development process helps clarify requirements*
- 25% of requirements change during development
  - *these account for 70%-85% of amount of rework*

Software Architecture

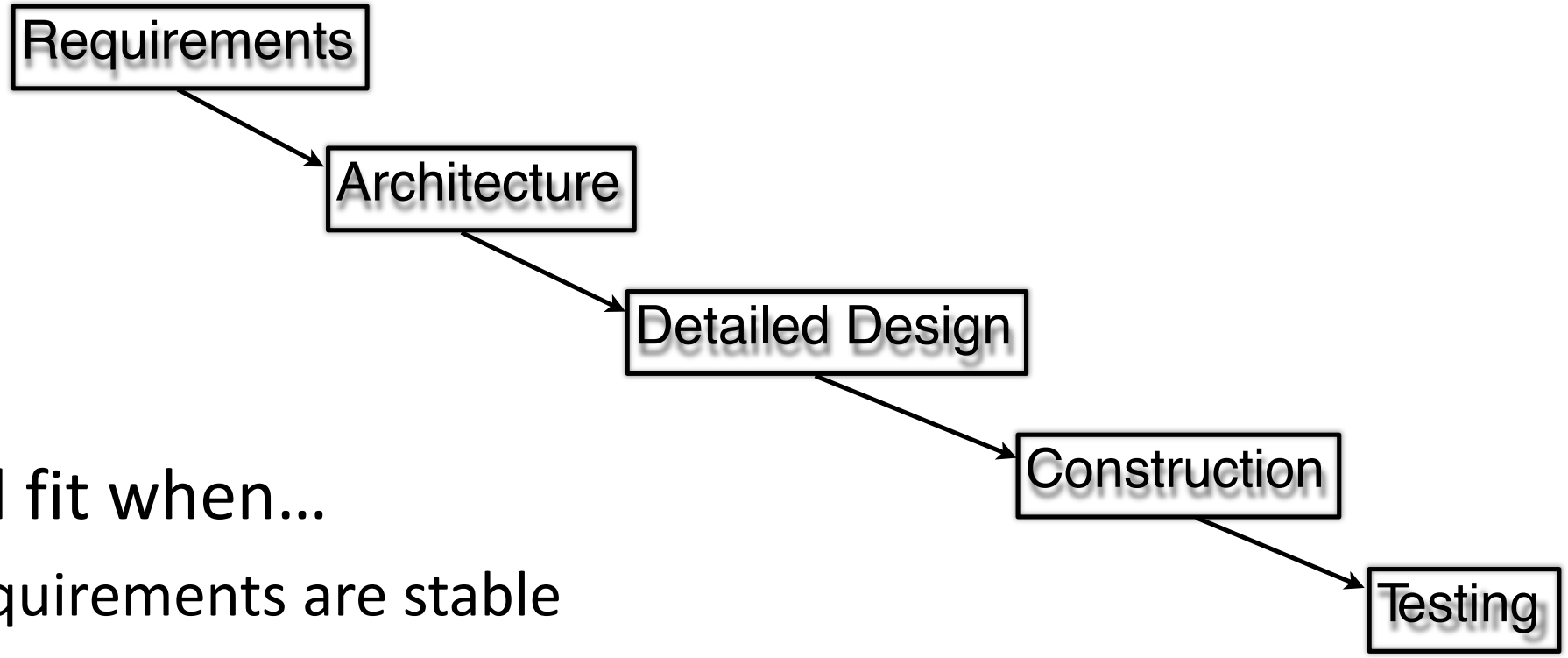
- Data design
  - *describe main data structures and files/DB tables*
  - *interoperability with other systems*
- User interface
  - *Web pages, GUI, CLI that enable user tasks from requirements*
  - *keep UI replaceable (useful for evolution, testing, etc.)*
- Resources
  - *estimates of maximum and average needs (e.g., for memory, disk space, threads, connections, network bandwidth, ...)*

Software Architecture

- What level of fault tolerance (1 fault, 2 faults, ...) ?
- Detection + handling + containment of errors/exceptions
- Build vs. reuse
  - *identify all reusable libs (GUI controls, communication, ...)*
- Internationalization
- Design for change / extensibility

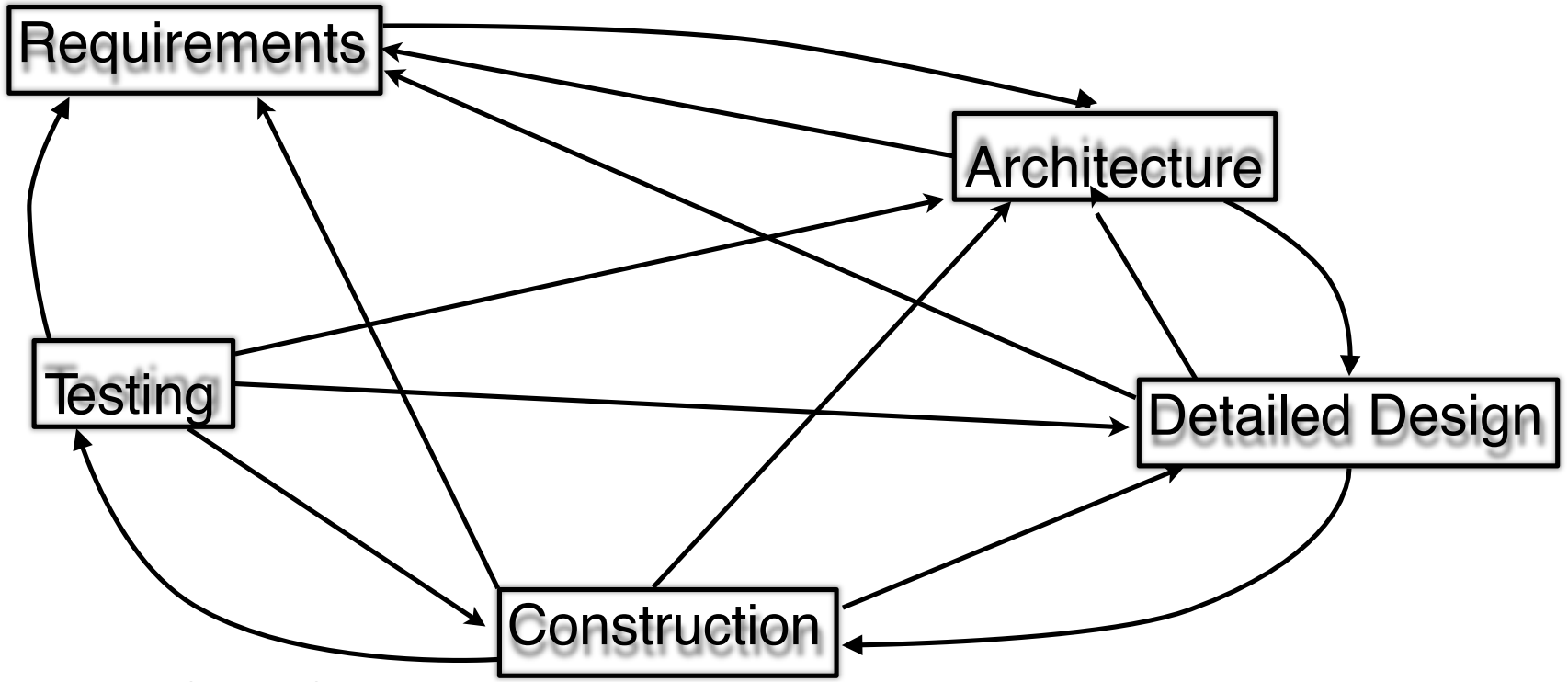

Future
Improvements
System testing
Construction
Architecture
Requirements
Problem Definition

Block Approach



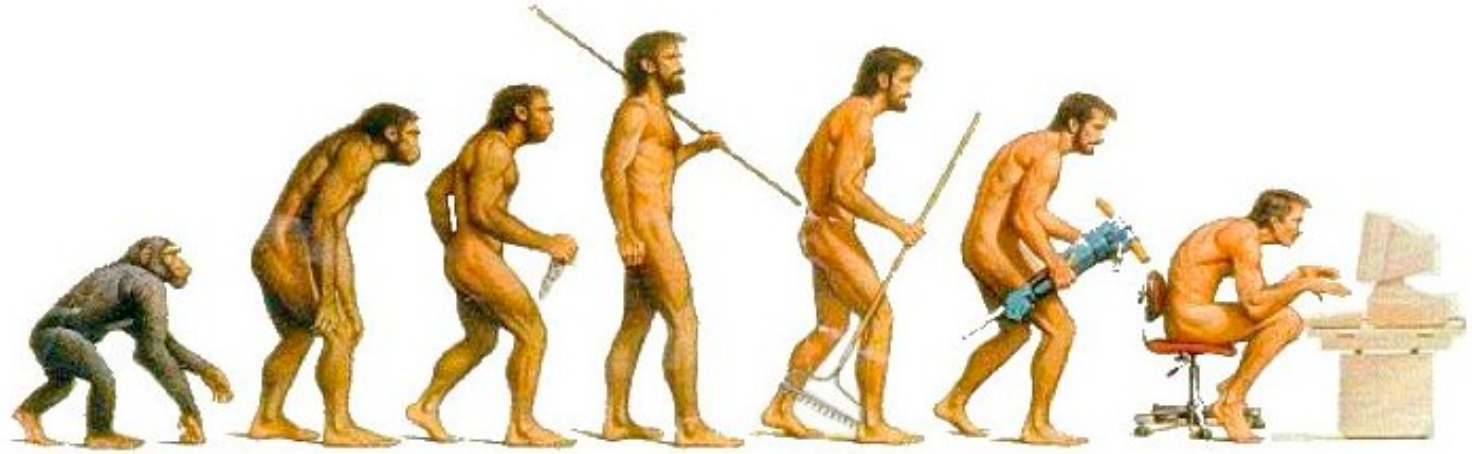
- Good fit when…
  - Requirements are stable
  - Design is straightforward and well understood
  - Development team need not “experiment”

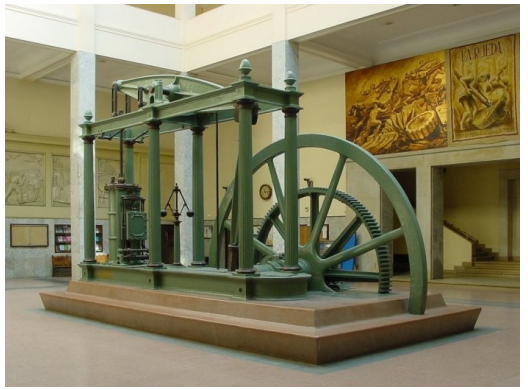
Agile Approach



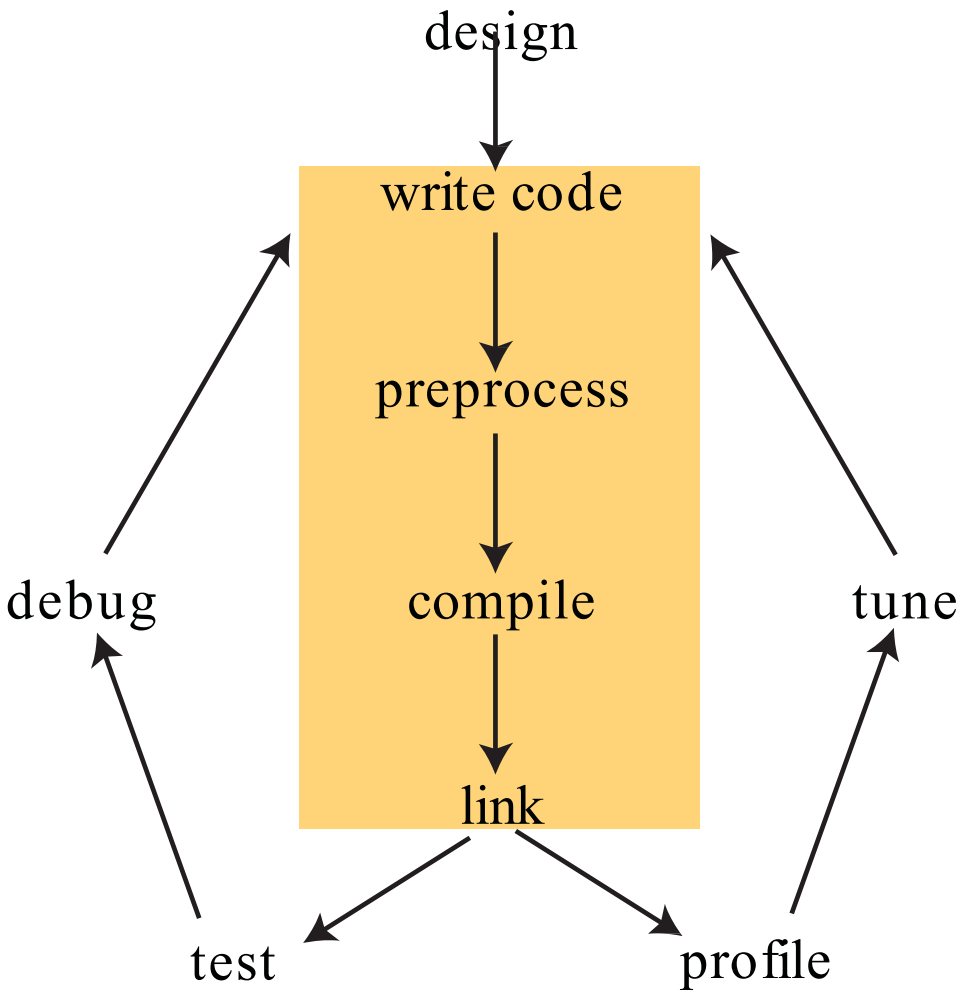
- Good fit when…
  - Requirements are not fully fleshed out
  - Design is complex and challenging
  - Development team not familiar with the class of software being built

The Software Developer’s Toolbox



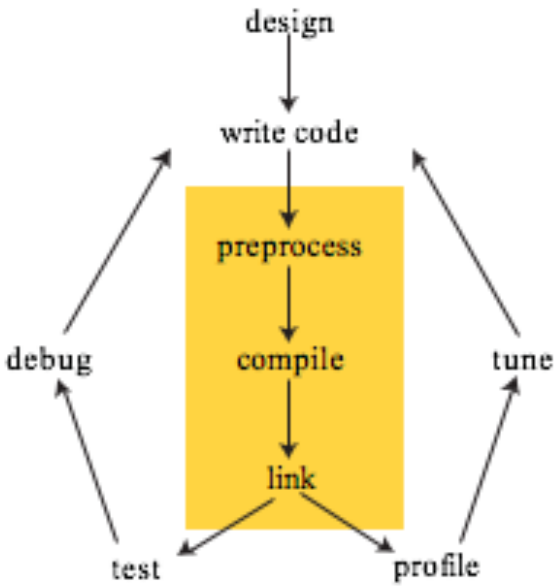


- Humanity progresses(?) when it gets new tools
- Always choose the right tools for the job
- Keep them sharp
  - Leading-edge tools -> coding productivity improves > 50%


design
write code
preprocess
compile
link
debug
tune
test
profile

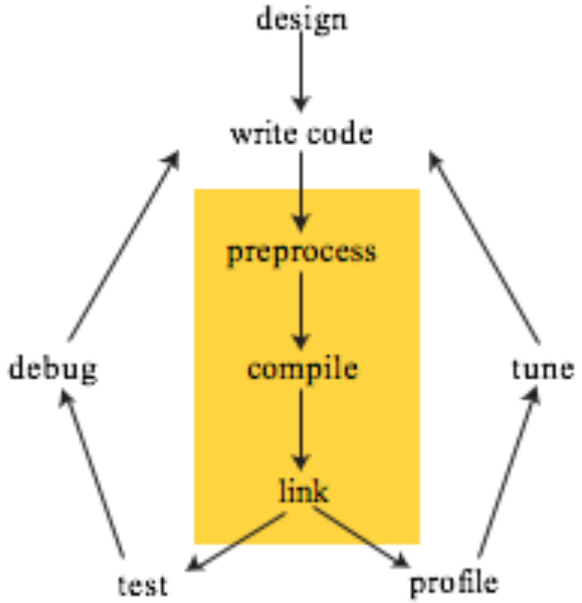
Build Tools

- Preprocessor
  - *macros → save typing, provide consistency*
  - *helps exclude debug code from shipped code*
- Compiler
  - *preprocessed source code → object code*
  - *static analyses (syntax, semantics, warnings)*
  - *can compute complexity metrics*
- Linker (static or dynamic)
  - *pieces together object code into executable software*



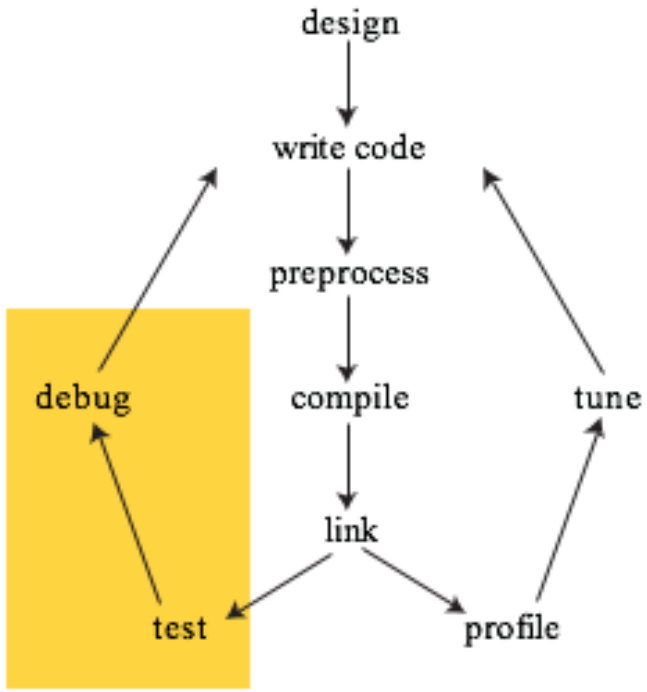
Build Tools

- Builders
  - *examples: maven, make*
  - *declaratively specify how to build the software*
  - *minimizes the time needed to build the executable*



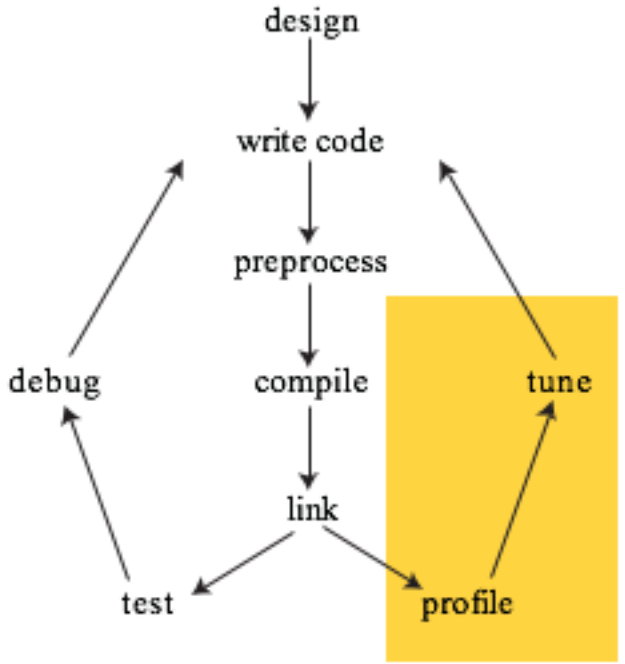
Testing/Debugging

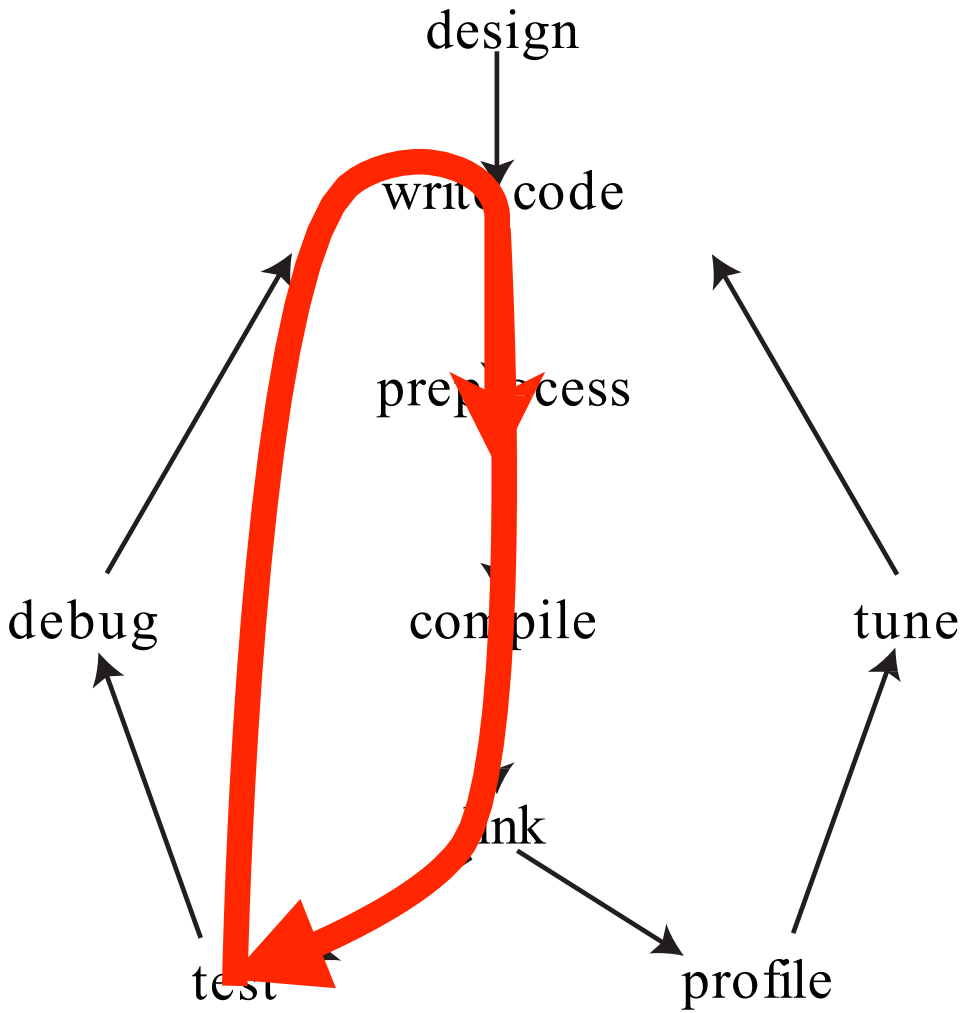
- Test framework
  - *unit test (JUnit, CppUnit, etc.)*
  - *test generators, record/playback tools*
  - *fault injectors*
  - *coverage tools*
- Debugging
  - *logging frameworks, trace monitors, diff tools*
  - *interactive debuggers*



Profiling

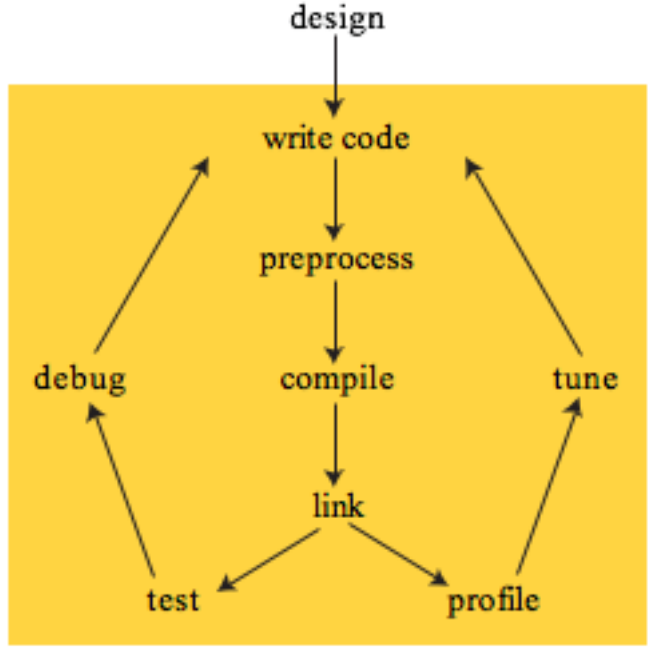
- Profiler is like a stethoscope
  - *count # execs of statements, functions, etc.*
  - *time spent, identify bottlenecks*
  - *trace library calls, system calls, etc.*




design
write code
preprocess
compile
link
test
profile
debug
tune

Development Environment

- Integrated Development Environments (IDEs)
  - *single portal into development sequence*
  - *edit code, refactor, navigate the code*
  - *auto-formatting and coloring code*
  - *compile, analyze, find errors, interactive help*
  - *templates, search-and-replace*



Collaboration Tools

- Version control system (VCS)
  - *ideally integrated with defect tracking*
  - *standardized configurations*
  - *make backups (and test your backup plan)*

More Tools

- Continuous integration servers (e.g., Jenkins)
- Documentation automators (Javadoc, Doxygen, ...)
- CASE (Computer-Aided Software Engineering)
  - *use visual representations to describe program logic*
  - *code generators (beware of maintainability)*