The Challenge of Parallelism

**Intel: Larrabee**

32 processors each 16-wide vector unit

**Nvidia: Fermi**

16 processors each 32-wide vector unit

Programming highly parallel processors is the software challenge of our era.
Outline

- What doesn’t work
  - Pieces of the problem ... and solution
  - General approach to architecting parallel sw
  - Detail on Structural Patterns
  - Detail on Computational Patterns
  - High-level examples of architecting applications

Assumption #1:
How not to develop parallel code

Initial Code → Profiler → Performance profile → Re-code with more threads → Not fast enough → Fast enough → Ship it → Lots of failures → N PE’s slower than 1
### Steiner Tree Construction Time By Routing Each Net in Parallel

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Serial</th>
<th>2 Threads</th>
<th>3 Threads</th>
<th>4 Threads</th>
<th>5 Threads</th>
<th>6 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>adaptec1</td>
<td>1.68</td>
<td>1.68</td>
<td>1.70</td>
<td>1.69</td>
<td>1.69</td>
<td>1.69</td>
</tr>
<tr>
<td>newblue1</td>
<td>1.80</td>
<td>1.80</td>
<td>1.81</td>
<td>1.81</td>
<td>1.81</td>
<td>1.82</td>
</tr>
<tr>
<td>newblue2</td>
<td>2.60</td>
<td>2.60</td>
<td>2.62</td>
<td>2.62</td>
<td>2.62</td>
<td>2.61</td>
</tr>
<tr>
<td>adaptec2</td>
<td>1.87</td>
<td>1.86</td>
<td>1.87</td>
<td>1.88</td>
<td>1.88</td>
<td>1.88</td>
</tr>
<tr>
<td>adaptec3</td>
<td>3.32</td>
<td>3.33</td>
<td>3.34</td>
<td>3.34</td>
<td>3.34</td>
<td>3.34</td>
</tr>
<tr>
<td>adaptec4</td>
<td>3.20</td>
<td>3.20</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
<td>3.21</td>
</tr>
<tr>
<td>adaptec5</td>
<td>4.91</td>
<td>4.90</td>
<td>4.92</td>
<td>4.92</td>
<td>4.92</td>
<td>4.92</td>
</tr>
<tr>
<td>newblue3</td>
<td>2.54</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
<td>2.55</td>
</tr>
<tr>
<td>average</td>
<td>1.00</td>
<td>1.0011</td>
<td>1.0044</td>
<td>1.0049</td>
<td>1.0046</td>
<td>1.0046</td>
</tr>
</tbody>
</table>

*Hint: What is this person thinking of?*

Re-code with more threads

Edward Lee, “The Problem with Threads”

Threads, locks, semaphores, data races
Outline

- What doesn’t work
- Pieces of the problem … and solution
- General approach to architecting parallel sw
- Detail on Structural Patterns
- Detail on Computational Patterns
- High-level examples of architecting applications

Building software: where we begin

Grady Booch
OO Guru

Can be built by one person
Requires
Minimal modeling
Simple process
Simple tools
The progress of Object Oriented Programming

Built most efficiently and timely by a team
Requires
- Modeling
- Well-defined process
- Power tools

Grady Booch
OO Guru

Goal - Future sw architecture

Progress
- Advances in materials
- Advances in analysis

Scale
- 5 times the span of the Pantheon
- 3 times the height of Cheops

Grady Booch
OO Guru
But … is a program like a building?

How is software like a building?  How is software NOT like a building?

Modularity is important …. But

Pop quiz: Is software more like?

a) A building  b) A factory
Object-Oriented Programming

Focused on:
- Program modularity
- Data locality
- Architectural styles
- Design patterns

Neglected:
- Application concurrency
- Computational details
- Parallel implementations

Modularity and locality have proved to be essential concepts for:
- Design
- Implementation
- Verification/test

What computations we do is as important than how we do them
High performance computing

HPC knows a lot about application concurrency, efficient programming, and parallel implementation

\[ x = \text{WS}_k \{W^*x\} \]
\[ x = F(P^T y + P^T \sigma, F^* x) \]

minimize \( ||Wx||_1 \)
\[ \text{s.t. } F_Gx = y, \]
\[ ||Gx - x||_2 < \varepsilon \]

\[ J(u) = \int_{\Omega} \psi_1((I(x + w) - I(x))^2)dx + \]
\[ \gamma \int_{\Omega} \psi_2((\nabla I(x + w) - \nabla I(x))^2)dx + \]
\[ \alpha \int_{\Omega} \psi_3((|\nabla u|^2 + |\nabla v|^2)dx \]

HPC approach to sw architecture

Technically this is known as a monolithic architecture
What’s a better metaphor for software development?

What we need

- Need to integrate the insights into computation provided by HPC with the insights into program structure provided by software architectural styles.

Software architecture

Computational patterns
- Linear Algebra
- Spectral
- Stencil

Structural patterns
- Graph algorithms
- FSMs
Alexander’s Pattern Language

Christopher Alexander’s approach to (civil) architecture:
- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." Page x, A Pattern Language, Christopher Alexander

Alexander’s 253 (civil) architectural patterns range from the creation of cities (2. distribution of towns) to particular building problems (232. roof cap)

A pattern language is an organized way of tackling an architectural problem using patterns

Main limitation:
- It’s about civil not software architecture!!!

Architecting Parallel Software with Patterns

Decompose Tasks/Data
Order tasks Identify Data Sharing and Access

Identify the Software Structure
- Pipe-and-Filter
- Agent-and-Repository
- Event-based
- Process Control
- Layered Systems
- Model-view controller
- Iterator
- MapReduce
- Arbitrary Task Graphs
- Puppeteer

Identify the Key Computations
- Graph Algorithms
- Dynamic programming
- Dense/Sparse Linear Algebra
- (Un)Structured Grids
- Graphical Models
- Finite State Machines
- Backtrack Branch-and-Bound
- N-Body Methods
- Circuits
- Spectral Methods
Outline

- What doesn’t work
- Pieces of the problem … and solution
- General approach to architecting parallel sw
  - Detail on Structural Patterns
  - Detail on Computational Patterns
  - High-level examples of architecting applications

Architecting Parallel Software

Decompose Tasks
  - Group tasks
  - Order Tasks

Decompose Data
  - Identify data sharing
  - Identify data access

Identify the Software Structure

Identify the Key Computations
Identify the SW Structure

Structural Patterns

- Pipe-and-Filter
- Agent-and-Repository
- Event-based coordination
- Iterator
- MapReduce
- Process Control
- Layered Systems

These define the structure of our software but they do not describe what is computed.

Analogy: Layout of Factory Plant
Identify Key Computations

Computational patterns describe the key computations but not how they are implemented.

Analogy: Machinery of the Factory
Analogy: Architected Factory

Raises appropriate issues like scheduling, latency, throughput, workflow, resource management, capacity etc.

Architecting Parallel Software with Patterns

<table>
<thead>
<tr>
<th>Decompose Tasks/Data</th>
<th>Identify Data Sharing and Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order tasks</td>
<td>Identify the Key Computations</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Identify the Software Structure
- Pipe-and-Filter
- Agent-and-Repository
- Event-based
- Bulk Synchronous
- MapReduce
- Layered Systems
- Arbitrary Task Graphs

Identify the Key Computations
- Graph Algorithms
- Dynamic programming
- Dense/Sparse Linear Algebra
- (Un)Structured Grids
- Graphical Models
- Finite State Machines
- Backtrack Branch-and-Bound
- N-Body Methods
- Circuits
- Spectral Methods
Outline

- What doesn’t work
- Pieces of the problem … and solution
- General approach to architecting parallel sw
- Detail on Structural Patterns
- Detail on Computational Patterns
- High-level examples of architecting applications

Inventory of Structural Patterns

1. pipe and filter
2. iterator
3. MapReduce
4. blackboard/agent and repository
5. process control
6. Model view controller
7. layered
8. event-based coordination
9. puppeteer
Elements of a structural pattern

- Components are where the computation happens
- A configuration is a graph of components (vertices) and connectors (edges)
- A structural patterns may be described as a family of graphs.

Connectors are where the communication happens

Pattern 1: Pipe and Filter

- Filters embody computation
- Only see inputs and produce outputs
- Pipes embody communication
- May have feedback

Examples?
Examples of pipe and filter

Almost every large software program has a pipe and filter structure at the highest level.

Pattern 2: Iterator Pattern

Initialization condition

 Variety of functions performed asynchronously

Synchronize results of iteration

Exit condition met?

Yes

No

Examples?
Example of Iterator Pattern: Training a Classifier: SVM Training

Pattern 3: MapReduce

To us, it means:
- A map stage, where data is mapped onto independent computations
- A reduce stage, where the results of the map stage are summarized (i.e. reduced)

Examples?
Examples of Map Reduce

- General structure:
  - Map a computation across distributed data sets
  - Reduce the results to find the best/(worst), maxima/(minima)

- Examples of Map Reduce
  - Support-vector machines (ML)
    - Map to evaluate distance from the frontier
    - Reduce to find the greatest outlier from the frontier
  - Speech recognition
    - Map HMM computation to evaluate word match
    - Reduce to find the most-likely word sequences

Pattern 4: Agent and Repository

- Agent and repository: Blackboard structural pattern
  - Agents cooperate on a shared medium to produce a result
  - Key elements:
    - **Blackboard**: repository of the resulting creation that is shared by all agents (circuit database)
    - **Agents**: intelligent agents that will act on blackboard (optimizations)
    - **Manager**: orchestrates agents access to the blackboard and creation of the aggregate results (scheduler)
Example: Compiler Optimization

- Constant folding
- Common-sub-expression elimination
- Loop fusion
- Strength-reduction
- Software pipelining
- Dead-code elimination

Optimization of a software program
- Intermediate representation of program is stored in the repository
- Individual agents have heuristics to optimize the program
- Manager orchestrates the access of the optimization agents to the program in the repository
- Resulting program is left in the repository

Example: Logic Optimization

- Timing opt agent 1
- Timing opt agent 2
- Timing opt agent 3
- Timing opt agent N

Optimization of integrated circuits
- Integrated circuit is stored in the repository
- Individual agents have heuristics to optimize the circuitry of an integrated circuit
- Manager orchestrates the access of the optimization agents to the circuit repository
- Resulting optimized circuit is left in the repository
Pattern 5: Process Control

Process control:
- **Process**: underlying phenomena to be controlled/computed
- **Actuator**: task(s) affecting the process
- **Sensor**: task(s) which analyze the state of the process
- **Controller**: task which determines what actuators should be effected

**Examples?:**

Examples of Process Control

Source: Adapted from Shaw & Garlan 1996, p27-31.
Pattern 9: Puppeteer

- Need an efficient way to manage and control the interaction of multiple simulators/computational agents
- Puppeteer Pattern – guides the interaction between the tasks/puppets to guarantee correctness of the overall task
- Puppeteer: 1) schedules puppets 2) manages exchange of data between puppets
- Difference with agent and repository?
  - No central repository
  - Data transfer between tasks/puppets

Examples?

Video Game
Model of circulation

- Modeling of blood moving in blood vessels
- The computation is structured as a controlled interaction between solid (blood vessel) and fluid (blood) simulation codes
- The two simulations use different data structures and the number of iterations for each simulation code varies
- Need an efficient way to manage and control the interaction of the two codes

Outline

- What doesn’t work
- Pieces of the problem … and solution
- General approach to architecting parallel sw
- Detail on Structural Patterns
- Detail on Computational Patterns
- High-level examples of architecting applications
Logic Optimization

Original Circuit

High Switching Frequency Net

Minimal Area Solution

Minimal Power Solution

Architecting Parallel Software

Decompose Tasks
• Group tasks
• Order Tasks
Decompose Data
• Data sharing
• Data access

Identify the Software Structure
Identify the Key Computations
Structure of Logic Optimization

Highest level structure is the pipe-and-filter pattern

*just because it's obvious doesn't mean it's not worth stating!*

Structure of optimization

Agent and repository: Blackboard structural pattern

Key elements:

- **Blackboard**: repository of the resulting creation that is shared by all agents (circuit database)
- **Agents**: intelligent agents that will act on blackboard (optimizations)
- **Controller**: orchestrates agents access to the blackboard and creation of the aggregate results (scheduler)
Timing Optimization

While (! user_timing_constraint_met && power_in_budget){
    restructure_circuit(netlist);
    remap_gates(netlist);
    resize_gates(netlist);
    ret ime_gates(netlist);
    ....
    more optimizations ...
    ....
}

Structure of Timing Optimization
Parallelism in Logic Synthesis

Logic synthesis offers lots of easy coarse-grain parallelism:
- Run n scripts/recipes and choose the best
- For per-instance parallelism: General program structure offers modest amounts amount of parallelism:
  - We can pipeline (pipe-and-filter) scanning, parsing, database/datamodel building
  - We can decouple agents (e.g. power and timing) acting on the repository
  - We can decouple sensor (e.g. timing analysis) and actuator (e.g. timing optimization)
  - We can use programming patterns like graph traversal and branch-and-bound
- But how do we keep 128 processors busy?
Here’s a hint …

Key to Parallelizing Logic Optimization?

We must exploit the data parallelism inherent in a graph/netlist with >2,000,000 cells
Partition graphs/netlists into highly/completely independent modules
Even modest amount of synchronization (e.g. stitching together overlapped regions) will devastate performance due to Amdahl’s law
Repository manager must partition underlying circuit to allow many agents (timing, power, area optimizers) to operation on different partitions simultaneously.

Chips with >2M cells easily enable opportunities for manycore parallelism.

**Moral of the story**

- Architecting an application doesn't automatically make it parallel.
- Architecting an application brings to light where the parallelism most likely resides.
- Humans must still analyze the architecture to identify opportunities for parallelism.
- However, significantly more parallelism is identified in this way than if we worked bottom-up to identify local parallelism.
Speedups

<table>
<thead>
<tr>
<th>Application</th>
<th>Speedups</th>
</tr>
</thead>
<tbody>
<tr>
<td>MRI</td>
<td>100x</td>
</tr>
<tr>
<td>SVM-train</td>
<td>20x</td>
</tr>
<tr>
<td>SVM-classify</td>
<td>109x</td>
</tr>
<tr>
<td>Contour</td>
<td>130x</td>
</tr>
<tr>
<td>Object Recognition</td>
<td>80x</td>
</tr>
<tr>
<td>Poselet</td>
<td>20x</td>
</tr>
<tr>
<td>Optical Flow</td>
<td>32x</td>
</tr>
<tr>
<td>Speech</td>
<td>11x</td>
</tr>
<tr>
<td>Value-at-risk</td>
<td>60x</td>
</tr>
<tr>
<td>Option Pricing</td>
<td>25x</td>
</tr>
</tbody>
</table>

Today’s take away

Many approaches to parallelizing software are not working
- Profile and improve
- Swap in a new parallel programming language
- Rely on a super parallelizing compiler

My own experience has shown that a sound software architecture is the greatest single indicator of a software project’s success.

Software must be architected to achieve productivity, efficiency, and correctness
- SW architecture >> programming environments
  - >> programming languages
  - >> compilers and debuggers
  - (>>hardware architecture)

If we had understood how to architect sequential software, then parallelizing software would not have been such a challenge

Key to architecture (software or otherwise) is design patterns and a pattern language

At the highest level our pattern language has:
- Eight structural patterns
- Thirteen computational patterns

Yes, we really believe arbitrarily complex parallel software can be built just from these!
More examples

Architecting Speech Recognition

Recognitions Network

Pipe-and-filter

Signal Processing

Voice Input

Inference Engine

Active State Computation Steps

Pipe-and-filter

MapReduce

Beam Search Iterations

Iterator

Most Likely Word Sequence

Large Vocabulary Continuous Speech Recognition Poster: Chong, Yi
Work also to appear at Emerging Applications for Manycore Architecture
CBIR Application Framework

Image histograms are common to many feature extraction procedures, and are an important feature in their own right

- Agent and Repository: Each agent computes a local transform of the image, plus a local histogram.
- Results are combined in the repository, which contains the global histogram

The data dependent access patterns found when constructing histograms make them a natural fit for the agent and repository pattern
Train Classifier: SVM Training

Exercise Classifier: SVM Classification
Key Elements of Kurt’s SW Education

- AT&T Bell Laboratories: CAD researcher and programmer
  - Algorithms: D. Johnson, R. Tarjan
  - Programming Pearls: S C Johnson, K. Thompson, (Jon Bentley)
  - Developed useful software tools:
    - Plaid: programmable logic aid: used for developing 100’s of FPGA-based HW systems
    - CONES/DAGON: used for designing >30 application-specific integrated circuits
  - Synopsys: researcher → CTO (25 products, ~15 million lines of code, $750M annual revenue, top 20 SW companies)
  - Super programming: J-C Madre, Richard Rudell, Steve Tjiang
  - Software architecture: Randy Allen, Albert Wang
  - High-level Invariants: Randy Allen, Albert Wang
- Berkeley: teaching software engineering and Par Lab
  - Took the time to reflect on what I had learned:
  - Architectural styles: Garlan and Shaw
    - Design patterns: Gamma et al (aka Gang of Four), Mattson’s PLPP
    - A Pattern Language: Alexander, Mattson
    - Dwarfs: Par Lab Team

Assumption #2: This won’t help either

- Code in new cool language
- Profiler
  - Performance profile
    - Fast enough
    - Ship it
  - Re-code with cool language
  - Not fast enough

After 200 parallel languages where’s the light at the end of the tunnel?
Parallel Programming environments in the 90's

<table>
<thead>
<tr>
<th>ABCPL</th>
<th>CORRELATE</th>
<th>GLU</th>
<th>2M</th>
<th>Muntat</th>
<th>Parallelogram2</th>
<th>pC++</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCE</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>ACE++</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Active messages</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada88</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada89</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada90</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada91</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada92</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada93</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada94</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada95</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada96</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada97</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada98</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada99</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
<tr>
<td>Ada2000</td>
<td>CORRELATE</td>
<td>GLU</td>
<td>2M</td>
<td>Muntat</td>
<td>Parallelogram2</td>
<td>pC++</td>
</tr>
</tbody>
</table>

Assumption #3: Nor this

Initial Code

Super-compiler

Performance profile

Tune compiler

Not fast enough

Fast enough

Ship it

30 years of HPC research don't offer much hope
Automatic parallelization?

Aggressive techniques such as speculative multithreading help, but they are not enough.

Ave SPECint speedup of 8% ... will climb to ave. of 15% once their system is fully enabled.

There are no indications auto par. will radically improve any time soon.

Hence, I do not believe Auto-par will solve our problems.

Results for a simulated dual core platform configured as a main core and a core for speculative execution.

---

Reinvention of design?

- In 1418 the Santa Maria del Fiore stood without a dome.
- Brunelleschi won the competition to finish the dome.
- Construction of the dome without the support of flying buttresses seemed unthinkable.
Innovation in architecture

- After studying earlier Roman and Greek architecture, Brunelleschi drew on diverse architectural styles to arrive at a dome design that could stand independently.

Innovation in tools

- His construction of the dome design required the development of new tools for construction, as well as an early (the first?) use of architectural drawings (now lost).

Scaffolding for cupola

http://www.templejc.edu/dept/Art/ASmith/ARTS1304/Joel/ZoomSlide0010.html

Mechanism for raising materials

http://www.artist-biography.info/gallery/filippo_brunelleschi/67/
Innovation in use of building materials

- His construction of the dome design also required innovative use of building materials.

Herringbone pattern bricks

http://www.buildingstonemagazine.com/winter-06/art/dome8.jpg

Resulting Dome

Completed dome

http://www.duomofirenze.it/storia/cupola_eng.htm
The point?

- Challenges to design and build the dome of Santa Maria del Fiore showed underlying weaknesses of architectural understanding, tools, and use of materials.
- By analogy, parallelizing code should not have thrown us for such a loop. Our difficulties in facing the challenge of developing parallel software are a symptom of underlying weakness in our abilities to:
  - Architect software
  - Develop robust tools and frameworks
  - Re-use implementation approaches
- Time for a serious rethink of all of software design.

Executive Summary

1. Our challenge in parallelizing applications really reflects a deeper more pervasive problem about inability to develop software in general
   1. Corollary: Any highly-impactful solution to parallel programming should have significant impact on programming as a whole
2. Software must be architectured to achieve productivity, efficiency, and correctness
3. SW architecture >> programming environments
   1. >> programming languages
   2. >> compilers and debuggers
   3. (>>hardware architecture)
4. Key to architecture (software or otherwise) is design patterns and a pattern language
5. The desired pattern language should span the full range of design from application conceptualization to detailed software implementation
6. Resulting software design then uses a hierarchy of software frameworks for implementation
   1. Application frameworks for application (e.g. CAD) developers
   2. Programming frameworks for those who build the application frameworks
What I’ve learned (the hard way)

Software must be architected to achieve productivity, efficiency, and correctness
SW architecture >> programming environments
  - >> programming languages
  - >> compilers and debuggers
  - (>> hardware architecture)
Discussions with superprogrammers taught me:
  - Give me the right program structure/architecture I can use any programming language
  - Give me the wrong architecture and I’ll never get there
What I’ve learned when I had to teach this stuff at Berkeley:
  - Key to architecture (software or otherwise) is design patterns and a pattern language
Resulting software design then uses a hierarchy of software frameworks for implementation
  - Application frameworks for application (e.g. CAD) developers
  - Programming frameworks for those who build the application frameworks