12 Design ⇒ Software Architecture

- Introduction to Design Readings
  - [dIP72] Parnas David L., “On the criteria to be used in decomposing systems into modules,” CACM, Dec., 1972

- Software Architecture Readings

Modular Decomposition

- How to define the structure of a modular system?
  - A module is a well-defined component of a software system
  - A module is part of a system that provides a set of services to other modules

- What are desirable properties of a decomposition?
  - Cohesion
  - Coupling
  - Complexity
  - Correctness
  - Correspondence

- Strategies for decomposition
  - Information Hiding
  - Layering
Cohesion & Coupling

- Cohesion: The reason that elements are found together in a module
  - Ex: coincidental, temporal, functional, …
  - The details aren’t critical, but the intent is useful
  - During maintenance, one of the major structural degradations is in cohesion
- Coupling: Strength of interconnection between modules
  - Hierarchies are touted as a wonderful coupling structure, limiting interconnections
  - But don’t forget about composition, which requires some kind of coupling
- It’s easy to...
  - ...reduce coupling by calling a system a single module
  - ...increase cohesion by calling a system a single module

Coupling also degrades over time
- “I just need one function from that module…”

Unnecessary coupling hurts
- Propagates effects of changes more widely
- Harder to understand interfaces (interactions)
- Harder to understand the design
- Complicates managerial tasks
- Complicates or precludes reuse
Coupling

- No satisfactory measure of coupling
- Either across modules or across a system
- Cruickshank and Gaffney Coupling metric

\[
Coupling = \frac{\sum_{i=1}^{n} Z_i}{\sum_{j=1}^{n} M_j}
\]

where:
\[
Z_i = \frac{\sum_{j=1}^{m} M_{ij}}{n}
\]

- \( M_j \) = sum of the number of input and output items shared between components \( i \) & \( j \)
- \( Z_i \) = average number of input and output items shared over \( m \) components with component \( i \)
- \( n \) = number of components in the software product

Complexity

- Simpler designs are better, all else being equal
- No useful measures of design/program complexity exist
- Although there are dozens of such measures
- LOC seems to be the most reliable predictor when single domains are considered, e.g.
  - Data processing
  - Numerical processing
  - Symbolic processing e.g., compilers
  - Concurrent/distributed systems e.g., operating systems
Correctness

- Very difficult (we’ll come back to this briefly later in the course)
- Even if you “prove” modules are correct, how do you prove
  - Composition of the modules within and outside the system to be designed
  - System software that must interpret/compile and run the composed modules
  - Hardware on which the complied modules run, etc.
- Leveson and others have shown clearly that a system can fail even when each of the pieces work properly

Correspondence

- “Problem-program mapping”
- The way in which the design is associated with the requirements
- The idea is that the simpler the mapping, the easier it will be to accommodate change in the design when the requirements change
- See M. Jackson: problem frames
Functional decomposition

- Divide-and-conquer based on functions
  - input;
  - compute;
  - output
- Then proceed to decompose compute
- This is stepwise refinement (Wirth, 1971)
  - There is an enormous body of work in this area, including many formal calculi to support the approach
  - Closely related to proving programs correct
  - More effective in the face of stable requirements

Properties of a modular structure

- Almost all the literature focuses on logical structures in design, but physical structure plays a big role in practice
- Sharing
- Separating work assignments
- Degradation over time

To what degree do you consider your systems
- as having modules?
- as consisting of a set of files?
- a question of physical vs. logical structure of programs
  - In some languages/environments, they are one and the same, e.g., Smalltalk-80
- How to define the structure of a modular system?
  - A module is a well-defined component of a software system
  - A module is part of a system that provides a set of services to other modules
    - where services are computational elements that other modules may use

Why so little attention paid to this?
**Modules and relations**

- Let S be a set of modules
  \[ S = \{ M_1, M_2, \ldots, M_n \} \]
- A binary relation \( R \) on S is a subset of
  \[ R \subseteq S \times S \]
- If \( M_i \) and \( M_j \) are in S, \( <M_i, M_j> \in R \) can be written as
  \( M_i R M_j \)
- Transitive closure \( R^+ \) of \( R \)
  \[ M_i R^+ M_j \iff M_i R M_j \text{ or } \exists M_k \in S \text{ s.t. } M_i R M_k \text{ and } M_k R^+ M_j \]
- \( R \) is a hierarchy iff there are no two elements \( M_i, M_j \) s.t.
  \[ M_i R M_j \text{ and } M_j R M_i \]
- Relations can be represented as graphs; a hierarchy is a DAG (directed acyclic graph)

*we assume our relations to be irreflexive*

---

**The uses relation**

- A uses B; examples
  - A requires the correct operation of B
  - A can access the services exported by B through its interface
  - A depends on B to provide its services
    - example: A calls a routine exported by B
  - A is a client of B; B is a server
  - the correctness of A depends on the presence of a correct version of B
    - requires specification and implementation of A and the specification of B
- Criteria for uses (A, B)
  - A is essentially simpler because it uses B
  - B is not substantially more complex because it does not use A
  - There is a useful subset containing B but not A
  - There is no useful subset containing A but not B
The uses relation

- Uses should be a hierarchy
  - Hierarchy makes software easier to understand
  - Proceed from leaf nodes (who do not use others) upwards
  - They make software easier to build
  - They make software easier to test
- A non-hierarchical uses relation makes it difficult to produce useful subsets of a system -- Parnas
  - It also makes testing difficult
  - (What about upcalls?)
  - So, it is important to design the uses relation
- Can uses be mechanically computed?
- Uses not the same as invokes

The is_component_of Relation

- Used to describe a higher level module as constituted by a number of lower level modules
- A is_component_of B
  - B consists of several modules, of which one is A
  - B comprises A
- If $M_{S,i} = \{M_k | M_k \in S \land M_k \text{ is_component_of } M_i\}$
  then we say that $M_{S,i}$ implements $M_i$
- Careful recording of (hierarchical) uses and is_component_of relations supports design of program families
A graphical view of is_component_of

They are a hierarchy

Hierarchy

- Organizes the modular structure through levels of abstraction
- Each level defines an abstract (virtual) machine for the next level
  - level can be defined precisely
    - $M_i$ has level 0 if no $M_j$ exists s.t. $M_i \succcurlyeq M_j$
    - let $k$ be the maximum level of all nodes $M_j$
      s.t. $M_i \succcurlyeq M_j$ ... then $M_i$ has level $k+1$
Interface design

- Interface should not reveal what we expect may change later
  - It should not reveal unnecessary details
  - Interface acts as a firewall preventing access to hidden parts
- Prototyping
  - Once an interface is defined, implementation can be done
    - first quickly but inefficiently
    - then progressively turned into the final version
  - Initial version acts as a prototype that evolves into the final product

Interface vs. implementation

- To understand the nature of uses, we need to know what a used module exports through its interface
  - The client imports the resources that are exported by its servers
  - Modules implement the exported resources
  - Implementation is hidden to clients
- Clear distinction between interface and implementation is a key design principle
  - Supports separation of concerns
    - clients care about resources exported from servers
    - servers care about implementation
  - Interface acts as a contract between a module and its clients
Information hiding

- perhaps the most important intellectual tool developed to support software design; makes anticipation of change a centerpiece in decomposition into modules
- are OO & IH the same?
  - OO classes are chosen based on the domain of the problem (in most OO analysis approaches), not necessarily based on change, but they are obviously related (e.g., separating interface from implementation)
- Notkin’s IH “Central Premises”
  1. can effectively anticipate changes
  2. changing an implementation is the best change, since it’s isolated
  3. semantics of a module must remain unchanged when implementations are replaced
  4. one implementation can satisfy multiple clients
  5. information hiding can be recursively applied

Layering [Parnas 79]

- Focus on information hiding modules isn’t enough
- One may also consider abstract machines
  - In support of program families
    - Systems that have “so much in common that it pays to study their common aspects before looking at the aspects that differentiate them”
- Still focusing on anticipated change
- Lots of language support for information hiding modules
  - C++ classes, Ada packages, etc.
- We have essentially no language support for layering
  - Operating systems provide support, primarily for reasons of protection, not abstraction
  - Big performance cost to pay for “just” abstraction
Categories of modules

- Functional modules
  - traditional form of modularization
  - provide a procedural abstraction
  - encapsulate an algorithm
- Libraries
  - a group of related procedural abstractions
    - e.g., mathematical libraries
      - implemented by routines of programming languages
- Common pools of data
  - data shared by different modules
    - e.g., configuration constants
      - the COMMON FORTRAN construct

Abstract Modules

- Abstract objects
  - Objects manipulated via interface functions
  - Data structure hidden to clients
- Abstract data types
  - Many instances of abstract objects may be generated
Abstract objects

- Examples
  - Calculator of expressions expressed in Polish postfix form: \( a \cdot (b+c) \rightarrow abc+ \)
  - A stack where the values of operands are shifted until an operator (assume only binary operators) is encountered in the expression

Interface of the abstract object STACK
exports
procedure PUSH (VAL: in integer);
procedure POP_2 (VAL1, VAL2: out integer);

- How does the design anticipate change in type of expressions to be evaluated?
  - E.g., it does not adapt to unary operators

Abstract data types (ADTs)

- Example: stack ADT
module STACK_HANDLER
exports
  type STACK = ?;
  This is an abstract data-type module; the data structure is a secret hidden in the implementation part.
  procedure PUSH (S: in out STACK ; VAL: in integer);
  procedure POP (S: in out STACK ; VAL: out integer);
  function EMPTY (S: in STACK) : BOOLEAN;
  ...
end STACK_HANDLER

- ADTs correspond to Java and C++ classes & may also be implemented by Ada private types and Modula-2 opaque types
Abstract data types (ADTs)

- Another example: simulation of a gas station
  module FIFO_CARS
  uses CARS
  exports
    type QUEUE : ?;
    procedure ENQUEUE (Q: in out QUEUE ; C: in CARS);
    procedure DEQUEUE (Q: in out QUEUE ; C: out CARS);
    function IS_EMPTY (Q: in QUEUE) : BOOLEAN;
    function LENGTH (Q: in QUEUE) : NATURAL;
    procedure MERGE (Q1, Q2 : in QUEUE ; Q : out QUEUE);
  This is an abstract data-type module representing
  queues of cars, handled in a strict FIFO way;
  queues are not assignable or checkable for equality,
  since ":=" and ":=" are not exported.

  ...
  end FIFO_CARS

Software Architecture

- architecture of a system describes its gross structure
- illuminates the top level design decisions
  - how the system is composed of interacting parts
  - the main pathways of interaction
  - the key properties of the parts
- allows high-level analysis and critical appraisal
- a bridge between requirements and implementation
  - an abstract description of a system,
  - exposes certain properties, while hiding others.
- useful for:
  - Understanding
  - Reuse
  - Construction
  - Evolution
  - Analysis
  - Management
Roles of Software Architecture

- **Understanding**:
  - simplifies the understanding of large systems using an abstraction
  - constraints on system design
  - rationale

- **Construction**:
  - a partial blueprint for development: components and dependencies

- **Evolution**:
  - dimensions along which a system is expected to evolve
  - "load-bearing walls" -> ramifications of changes, cost estimation
  - separate concerns about the functionality of a component from the ways in which that component is connected to (interacts with) other components

- **Analysis**:
  - consistency checking
  - conformance
  - to constraints
  - to quality attributes
  - dependence analysis
  - domain-specific analyses for architectural styles

- **Reuse**
  - reuse of large components and frameworks

- **Management**
  - leads to a much clearer understanding of requirements, implementation strategies, and potential risks

Software Architectures

- Architectural taxonomy ("boxology")
- Architectural patterns & idioms
- Design patterns & idioms
- Reuse
  - Class libraries
  - Components
  - Frameworks
  - Middleware
Frameworks & Class Libraries

- A framework is an integrated set of abstract classes that can be customized for instances of a family of applications.
- A class is a unit of abstraction & implementation in an OO programming language.
- A component is an encapsulation unit with one or more interfaces that provide clients with access to its services.

Comparison

<table>
<thead>
<tr>
<th>Class Libraries</th>
<th>Frameworks</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro-level</td>
<td>Meso-level</td>
<td>Macro-level</td>
</tr>
<tr>
<td>Stand-alone language entities</td>
<td>“Semi-complete” applications</td>
<td>Stand-alone composition entities</td>
</tr>
<tr>
<td>Domain-independent</td>
<td>Domain-specific</td>
<td>Domain-specific or Domain-independent</td>
</tr>
<tr>
<td>Borrow caller’s thread</td>
<td>Inversion of control</td>
<td>Borrow caller’s thread</td>
</tr>
</tbody>
</table>

Adapted from Douglas C. Schmidt, “Patterns, Frameworks & Middleware: Their Synergistic Relationships.”
Historically

- Architecture was largely ad hoc affair
  - Designers freely use informal patterns/idioms
    - informal with imprecise semantics
    - diagrams + prose, but no rules
  - Designers use system-level abstraction
    - overall organization (styles)
    - components and interactions
  - Designers compose systems from subsystems
    - but, tend to think statically
    - select structure by default, rather than by design
- Key events
  - Parnas recognized the importance of system families and architectural decomposition principles based on information hiding
  - Dijkstra proposed certain system structuring principles

Architecture was largely ad hoc

- Control Process (CP)
- Prop Loss Model (MODP)
- Reverb Model (MODR)
- Noise Model (MODN)

- is this an architecture?
Example

- what is the nature of the components, and what is the significance of their separation?

Example

- what is the significance of the links?

- what is the significance of the layout?
Abstraction techniques in CS

- Programming Languages
  - machine language
  - symbolic assemblers
  - macro processors
  - early high-level languages
    - Fortran
      - data types served primarily as cues for selecting the proper machine instructions
    - Algol and its successors
      - data types serve to state the programmer’s intentions about how data should be used.
  - later high-level languages
    - separation of a module’s specification from its implementation
    - introduction of abstract data types.

- ADT
  - the software structure (which included a representation packaged with its primitive operators)
  - specifications (mathematically expressed as abstract models or algebraic axioms)
  - language issues (modules, scope, user-defined types)
  - integrity of the result (invariants of data structures and protection from other manipulation)
  - rules for combining types (declarations)
  - information hiding (protection of properties not explicitly included in specifications)
two trends

- recognition of a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems
- concern with exploiting commonalities in specific domains to provide reusable frameworks for product families
two trends

- concern with exploiting commonalities in specific domains to provide reusable frameworks for product families; examples include:
  - the standard decomposition of a compiler
  - standardized communication protocols, e.g., Open Systems Interconnection Reference Model (a layered network architecture)
  - tools, e.g., NIST/ECMA Reference Model (a generic software engineering environment architecture based on layered communication substrates)
  - fourth-generation languages
  - user interface toolkits and frameworks, e.g., X Window System (a distributed windowed user interface architecture based on event triggering and callbacks)

Why Important?

- mutual communication.
  - software architecture represents a common high-level abstraction of the system that most, if not all, of the system’s stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.

- transferable abstraction of a system.
  - software architecture embodies a relatively small, intellectually graspable model for how the system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements, and can promote large scale reuse.
Why Important?

- early design decisions
  - software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system’s remaining development, its service in deployment, and its maintenance life.

- architecture
  - provides builders with constraints on implementation
  - dictates organizational structure for development and maintenance projects
  - permits or precludes the achievement of a system’s targeted quality attributes
  - helps in predicting certain qualities about a system architecture can be the basis for training
  - helps in reasoning about and managing change

Architecture=

- elements
  - processing
  - data
  - connectors

- form
  - rules which constrain element placement
  - style/design

- rationale
  - selection of form
  - links to reqmnts & design
  - functional/non-functional attributes

Elements, form, rationale, views

Process View

Data View
architectural styles/idioms

- architectural style =
  - Components: locus of computation
    - filters, databases, objects, clients, servers, ADTs
  - Connectors: mediate interactions of components
    - procedure call, pipes, event broadcast
  - Properties: specify info for construction & analysis
    - Signatures, pre/post conditions, RT specifications
- other
  - topology
  - underlying structural model?
  - underlying computational model?

Expected Benefits

- Requirements
  - Clarify intentions
  - Make decisions and implications explicit
  - Permit system level analysis
- Architecture
- Design
  - Code Integration
- Test Accept
- Maintenance
  - Reduce maintenance costs, directly and indirectly

© David Garlan CMU
Components and connectors
- primary building blocks of architectures
- abstractions used by designers in defining their architectures
- most of these elements are ultimately implemented in terms of processes (as defined by the operating system) and procedure calls (as defined by the programming language).

Control issues
- Topology
  - geometric form of the control flow for the system: linear (non-branching), acyclic, hierarchical, star, arbitrary
- Synchronicity
  - interdependency of the component control states: lockstep (sequential or parallel), synchronous, asynchronous, opportunistic
- Binding time
  - time the identity of a partner in a transfer-of-control operation is established: write (i.e., source code)

Data issues
- Mode
  - data is made available throughout the system: passed (object style from component to component), shared: copyout-copy-in, broadcast, multicast
- Binding time
  - time identity of a partner in a data operation is established: write (i.e., source code)

Control/data interaction issues
- Shape
  - control flow and data flow topologies isomorphic
- Directionality
  - if shapes the same, does control flow in the same direction as data or the opposite direction.

Type of reasoning
- nondeterministic state machine theory, function composition
- software substructure and analysis substructure should be compatible.
**taxonomy: data flow**

- **batch sequential**
  - independent programs, dataflow in large chunks, no parallelism

- **pipes & filters**
  - incremental, byte stream data flow, pipelined “parallelism”, local context, no state persistence

---

**Boxology: dataflow**

<table>
<thead>
<tr>
<th>Style</th>
<th>Constituent parts</th>
<th>Control issues</th>
<th>Data issues</th>
<th>Ctrl/data interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Components</td>
<td>Connectors</td>
<td>Topology</td>
<td>Synchr.icity</td>
</tr>
<tr>
<td>Dataflow network [B+88]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Acyclic [A+95]</td>
<td>data stream</td>
<td>arbitrary</td>
<td>arbitrary</td>
<td></td>
</tr>
<tr>
<td>• Fanout [A+95]</td>
<td>transducers</td>
<td>acyclic</td>
<td>acyclic</td>
<td>i, r</td>
</tr>
<tr>
<td>• Pipeline [DG+90, S+88, A+95]</td>
<td>ascii stream</td>
<td>hierarchical (async)</td>
<td>hierarchical (async)</td>
<td>i, r</td>
</tr>
<tr>
<td>- Unix pipes and filters [Bab86]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key to column entries**

- **Synchr.icity**: asynch (asynchronous), i (invocation-time), r (run-time)
- **Continuity**: cont (continuous), lvol (high-volume), lvol (low-volume)
Analysis: pipes & filters*

- problem decomposition
  - advantages: hierarchical decomposition of system function
  - disadvantages: “batch mentality,” interactive apps?, design

- maintenance & reuse
  - advantages: extensibility, reuse, “black box” approach
  - disadvantages: lowest common denominator for data flow

- performance
  - advantages: pipelined concurrency
  - disadvantages: parsing/un-parsing, queues, deadlock with limited buffers

*to some extent batch

Rules of thumb for dataflow/pipes

- If your problem can be decomposed into sequential stages, consider batch sequential or pipeline architectures
- If in addition each stage is incremental, so that later stages can begin before earlier stages complete, then consider a pipelined architecture
- If your problem involves transformations on continuous streams of data (or on very long streams) consider a pipeline architecture
  - However, if your problem involves passing rich data representation, then avoid pipeline architectures restricted to ASCII
- If your system involves controlling action, is embedded in a physical system, and is subject to unpredictable external perturbation so that preset algorithms go awry, consider a closed loop architecture
taxonomy: call/return

- main/sub
  - hierarchical decomposition, single thread of control, structure implicit, correctness depends on subordinates
- layered
  - hides lower layers/services higher layer, upper="virtual machines"/lower =hw, kernel, scoping
- object-oriented
  - encapsulation, inheritance, polymorphism

Analysis: call/return

- layers
  - portability, modifiability, reuse
    - advantages: each layer is abstract machine, each layer interacts with ≤2 other layers, standard interfaces
  - performance, design
    - disadvantages: semantic feedback in UI, deep functionality, abstractions difficult, bridging layers
- object-oriented
  - portability, modifiability, reuse
    - advantages: decreased coupling, frameworks -> reuse
  - performance, design
    - advantages: maps easily to "real world", inheritance, encapsulation
    - disadvantages: design harder, side effects, identity, inheritance difficult
**Taxonomy: data-centered**

- **transactional db**
  - large central data store, control via transactions

- **blackboards**
  - central shared + app-specific data representations, control via data state

---

**Rules of thumb: objects and repositories**

- If a central issue is understanding the data of the application, its management, and its representation, consider a repository or ADT architecture; if the data is long-lived focus on repositories
- If the representation of data is likely to change over the lifetime of the program, ADTs or objects can confine the changes to particular components
- If you are considering repositories and the input data is “noisy” and the execution order can not be predetermined, consider a blackboard
- If you are considering repositories and the execution order is determined by a stream of incoming requests and the data is highly structured, consider a DB system.
**Taxonomy: independent components**

- **communicating processes**
  - independent processes, point-point message passing, asynch/synch, RPC layered on top

- **event systems**
  - interface define allowable in/out events, event-procedure bindings: procedure "registration", communication by event "announcement", implicit action invocation on event, non-deterministic ordering

---

**Boxology: independent components**

<table>
<thead>
<tr>
<th>Style</th>
<th>Constituent parts</th>
<th>Control issues</th>
<th>Data issues</th>
<th>Ctrl/data interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Components</td>
<td>Connectors</td>
<td>Topology</td>
<td>Synchrony</td>
</tr>
<tr>
<td>Communicating processes</td>
<td>arb</td>
<td>arb</td>
<td>seq</td>
<td>arb</td>
</tr>
<tr>
<td>One-way data flow</td>
<td>linear</td>
<td>star</td>
<td>linear</td>
<td>star</td>
</tr>
<tr>
<td>Balanced</td>
<td>star</td>
<td>star</td>
<td>star</td>
<td>star</td>
</tr>
<tr>
<td>Heartbeat</td>
<td>processes</td>
<td>processes</td>
<td>processes</td>
<td>processes</td>
</tr>
<tr>
<td>Probe/echo</td>
<td>message</td>
<td>message</td>
<td>message</td>
<td>message</td>
</tr>
<tr>
<td>Broadcast</td>
<td>arb</td>
<td>arb</td>
<td>star</td>
<td>star</td>
</tr>
<tr>
<td>Dynamicassign</td>
<td>arb</td>
<td>arb</td>
<td>arb</td>
<td>arb</td>
</tr>
<tr>
<td>Decentralized</td>
<td>kier</td>
<td>kier</td>
<td>kier</td>
<td>kier</td>
</tr>
<tr>
<td>Replicated</td>
<td>message</td>
<td>message</td>
<td>message</td>
<td>message</td>
</tr>
<tr>
<td>workers</td>
<td>kier</td>
<td>kier</td>
<td>kier</td>
<td>kier</td>
</tr>
</tbody>
</table>

**Key to column entries**

- **Topology**
  - arb (asynchronous), arb (synchronous), star, linear (one-way)
  - seq (sequential), star (one-to-one), star (one-to-many), sync (synchronous), async (asynchronous), app (application)

- **Synchrony**
  - seq, synch, par, i (inter-processor), s (source-code), c (compile-time), i (inclusion-time)

- **Binding time**
  - ar, star, sync, par, i (inclusion-time), t (runtime)

- **Continuity**
  - arb (source), seq, par (source), i (inclusion), c (copy)

- **Mode**
  - shared, passed, broadcast, event (multicast), c/c (copy-to-copy-out)
analysis

- event systems
- portability, modifiability, reuse
  - advantages: no “hardwired names”, new objects added by registration
  - disadvantages: nameserver/“yellowpages” needed
- performance, design
  - advantages: computation & coordination are separate objects/more independent, parallel invocations
  - disadvantages: no control over order of invocation, correctness, performance penalty from communication overhead

Rules of thumb

- If your task requires a high degree of flexibility-configurability, loose coupling between tasks, and reactive tasks, consider interacting processes
  - If you have reason not to bind the recipients of signals to their originators, consider an event architecture
  - If the task are of a hierarchical nature, consider a replicated worker or heartbeat style
  - If the tasks are divided between producers and consumers, consider a client-server style (naive or sophisticated)
  - If it makes sense for all of the tasks to communicate with each other in a fully connected graph, consider a token-passing style
**taxonomy: virtual machines**

- **interpreters**
  - simulate functionality which is not native to the run-time system; execution engine “implemented” in software

- **rule-based systems**
  - specialization of an interpreter

---

**Analysis: virtual machines**

- **interpreters**
- portability, modifiability, reuse
  - disadvantages: map into actual implementation?

- **performance, design**
  - advantages: simulate non-native functionality, can simulate “disaster” modes for safety analysis
  - disadvantages: much slower than actual system, additional layer of software to be verified

- **Rules of thumb: virtual machines**
  - If you have designed a computation, but have no machine on which you can execute it, consider a virtual interpreter architecture.