Why Domain-Specific Software Architectures?

- Development in specific application domains can be optimized
- Reuse in specific domains is most realistic
  - reuse in general is too difficult to achieve
  - focus on particular classes of applications with similar characteristics
- Criteria for successful reuse [Biggerstaff]
  - well-understood domain
  - slowly changing
  - has intercomponent standards
  - provides economies of scale
  - fits existing infrastructure
Software Development

Problem Space

Solution Space
Architecture-Based Software Development

Problem Space

Solution Space
DSSA-Based Software Development

Problem Space

Solution Space
What Is DSSA?

- DSSA is an assemblage of **software components**
  - specialized for a particular type of task (domain)
  - **generalized** for effective use across that domain
  - composed in a **standardized structure** (topology) effective for building successful applications
    - *Rick Hayes-Roth, 1994*

- DSSA is comprised of
  - a **domain model**,
  - **reference requirements**,
  - a **reference architecture** (expressed in an ADL),
  - its supporting **infrastructure/environment**, and
  - a **process/methodology** to instantiate/refine and evaluate it.
    - *Will Tracz, 1995*
How Does One Leverage DSSAs?

- **Reference models**
  - specific configurations for certain application areas
  - may be effective outside their initial domains
    → discussed previously
      - e.g., canonical compiler architectures

- **Idioms & patterns**
  - deal with global organizational structures
  - application-domain independent
    → to be discussed today
      - pipe and filter
      - client-server
      - blackboard
      - layered
Definitions of Style

- Architectural styles are recurring organizational patterns and idioms.
  
  – Shaw & Garlan

- Established, shared understanding of common design forms is a mark of a mature engineering field.
  
  – Shaw & Garlan

- Architectural style is an abstraction of recurring composition and interaction characteristics of a set of architectures.
  
  – Taylor

- Styles are key design idioms that enable exploitation of suitable structural and evolution patterns and facilitate component, connector, and process reuse.
  
  – Medvidovic
Basic Properties of Styles

- A *vocabulary* of design elements
  - component and connector types
  - e.g., pipes, filters, objects, servers

- A set of *configuration rules*
  - topological constraints that determine allowed compositions of elements
  - e.g., a component may be connected to at most two other components

- A *semantic interpretation*
  - compositions of design elements have well-defined meanings

- Possible *analyses* of systems built in a style
  - code generation is a special kind of analysis
Benefits of Styles

- Design reuse
  - well-understood solutions applied to new problems
- Code reuse
  - shared implementations of invariant aspects of a style
- Understandability of system organization
  - a phrase such as “client-server” conveys a lot of information
- Interoperability
  - supported by style standardization
  - e.g., CORBA, JavaBeans
- Style-specific analyses
  - enabled by the constrained design space
- Visualizations
  - style-specific depictions matching engineers’ mental models
Style Analysis Dimensions

- What is the design vocabulary?
- What are the allowable structural patterns?
- What is the underlying computational model?
- What are the essential invariants of the style?
- What are common examples of its use?
- What are the (dis)advantages of using the style?
- What are the style’s specializations?
Some Common Architectural Styles

- Pipe and filter
- Object-oriented
- Layered
- Blackboard
- State transition
- Client-server
- Peer-to-peer
- Event-based (a.k.a. implicit invocation)
- Push-based
Pipe and Filter Style

- Components are filters
  - transform input data streams into output data streams
  - possibly incremental production of output

- Connectors are pipes
  - conduits for data streams

- Style invariants
  - filters are independent
  - a filter has no knowledge of up- and down-stream filters

- Examples
  - UNIX shell
  - distributed systems
  - signal processing
  - parallel programming
Pipe and Filter (cont.)

- Variations
  - pipelines — linear sequences of filters
  - bounded pipes — limited amount of data on a pipe
  - typed pipes — data strongly typed
  - batch sequential — data streams are not incremental

- Advantages
  - \( \text{system.behavior} = \sum_i \text{component}_i.\text{behavior} \)
  - filter addition, replacement, and reuse
  - certain analyses
  - concurrent execution

- Disadvantages
  - batch organization of processing
  - interactive applications
  - lowest common denominator on data transmission
Object-Oriented Style

- Components are objects
  - data and associated operations
- Connectors are messages and method invocations
- Style invariants
  - objects are responsible for their internal representation integrity
  - internal representation is hidden from other objects
- Advantages
  - “infinite malleability” of object internals
  - system decomposition into sets of interacting agents
- Disadvantages
  - objects must know each other’s identities
  - side effects in object method invocations
Layered Style

- Hierarchical system organization
  - “multi-level client-server”
  - each layer exports an “API” to be used by above layers
- Each layer acts as a
  - server: service provider to layers “above”
  - client: service consumer from layers “below”
- Connectors are protocols of layer interaction
- Example — operating systems
- Virtual machine style results from fully opaque layers
Layered Style (cont.)

- Advantages
  - increasing abstraction levels
  - evolvability
  - reuse

- Disadvantages
  - not universally applicable
  - performance
  - determining the correct abstraction level
Blackboard Style

- Two kinds of components
  - central data structure — blackboard
  - components operating on the blackboard
- System control is entirely driven by the blackboard state
- Examples
  - typically used for AI systems
  - integrated software environments (e.g., Interlisp)
  - compiler architecture
State-Transition Style

- Components represent (sets of) system states
- Connectors are (sets of) named state transitions

- Disadvantage
  - even trivial systems have enormous state spaces

→ Remedy
  - abstract away states into coarser-grained components
  - e.g., StateCharts/StateMate
Client-Server Style

- An instance of a more general style
  → distributed systems
- Components are clients and servers
- Servers do not know the number or identities of clients
- Clients know server’s identity
- Connectors are RPC-based interaction protocols
- A number of different flavors of client-server
Implicit Invocation Style

- Event announcement instead of method invocation
  - “listeners” register interest in and associate methods with events
  - the system invokes all registered methods implicitly
- Component interfaces are methods and events
- Two types of connectors
  - invocation is either explicit or implicit in response to events
- Style invariants
  - “announcers” are unaware of their events’ effects, if any
  - no assumption about processing in response to events
Implicit Invocation (cont.)

- Advantages
  - component reuse
  - system evolution
    - both at system construction-time and run-time

- Disadvantages
  - counter-intuitive system structure
  - components relinquish computation control to the system
  - no knowledge of what component(s) will respond to an event
  - no knowledge of the order of responses
  - analysis via pre- and post-conditions is difficult
Push-Based Style

- Distinguished from pull-based (e.g., the Web)

  ![Diagram of consumer and producer with interaction arrows]

- Examples
  - employee information systems
  - maintenance manuals
  - stock ticker
Push-Based Style (cont.)

- **Components**
  - producer
  - receiver

- **Connectors**
  - channel
  - broadcaster
  - transport system
    - repeater, cache, proxy

- **Asymmetric communication model**
  - producers $\neq$ receivers
  - fewer producers but more receivers per producer than event-based style

- Relatively tight coupling between source and receiver via subscribed channels
Heterogeneous Styles

- **FIFO Queue**
- **Proc Calls**
- **Subscribe**
- **Push**
- **Blackboard**
Observations

- Different styles result in
  - different architectures
  - architectures with greatly differing properties

- A style does not fully influence the resulting architecture
  - a single style can result in different architectures
  - considerable room for individual judgement
  - variations among architects
  - different emphases
    - e.g., imposed by the customer

- A style defines a domain of discourse
  - about a problem (domain)
  - about the resulting system
  - different architectures lead architects to ask different questions