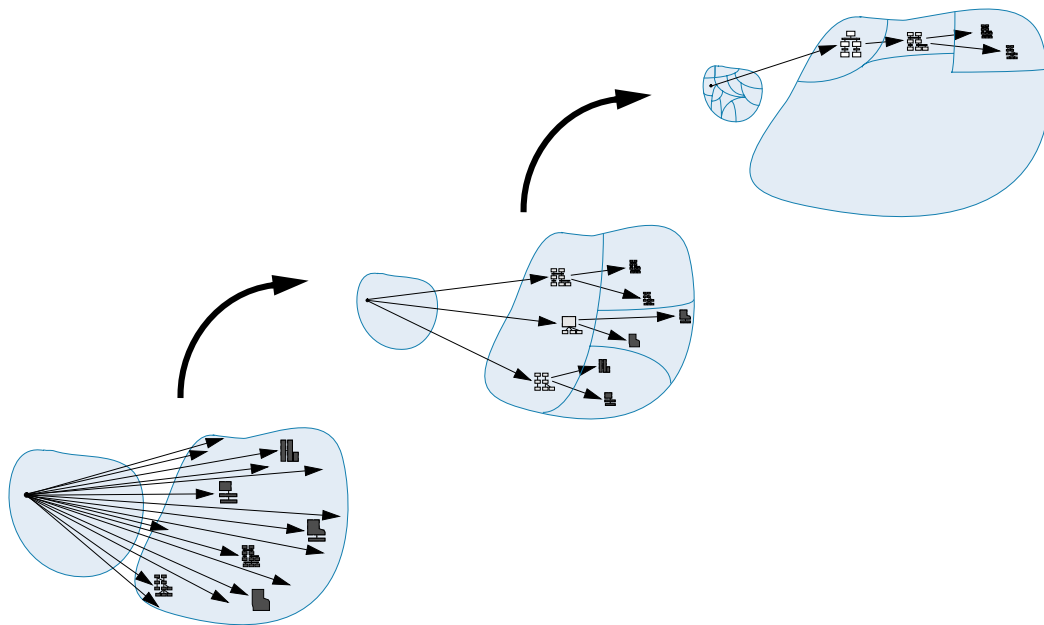
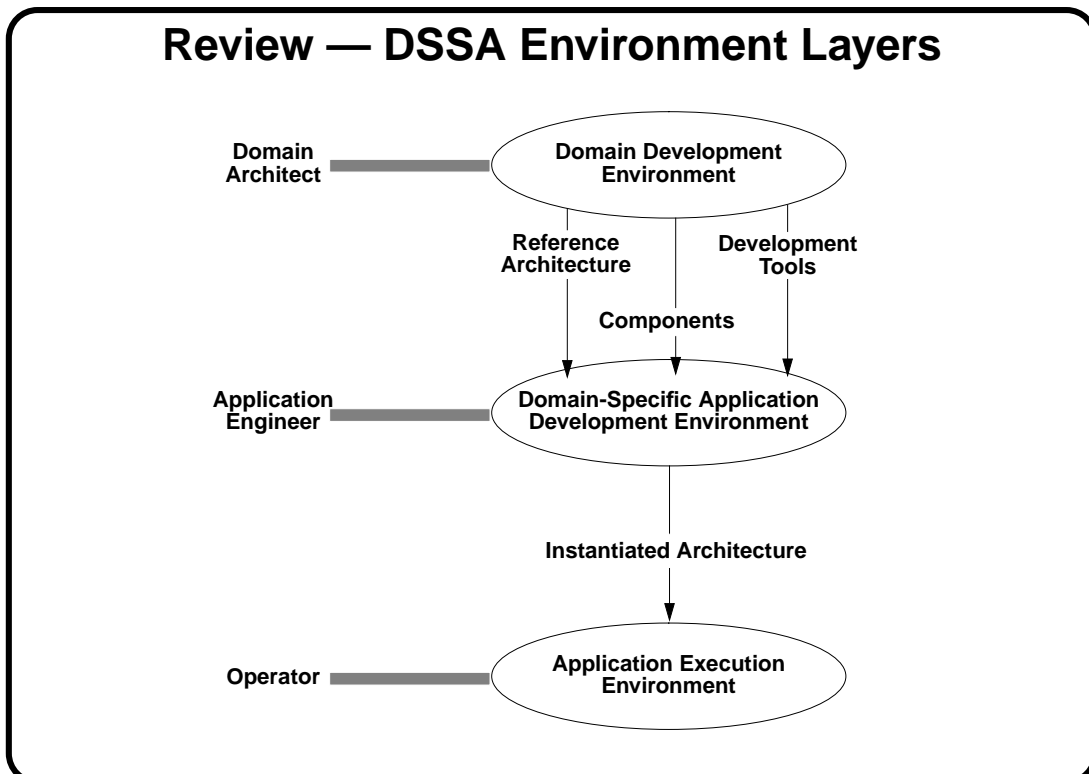
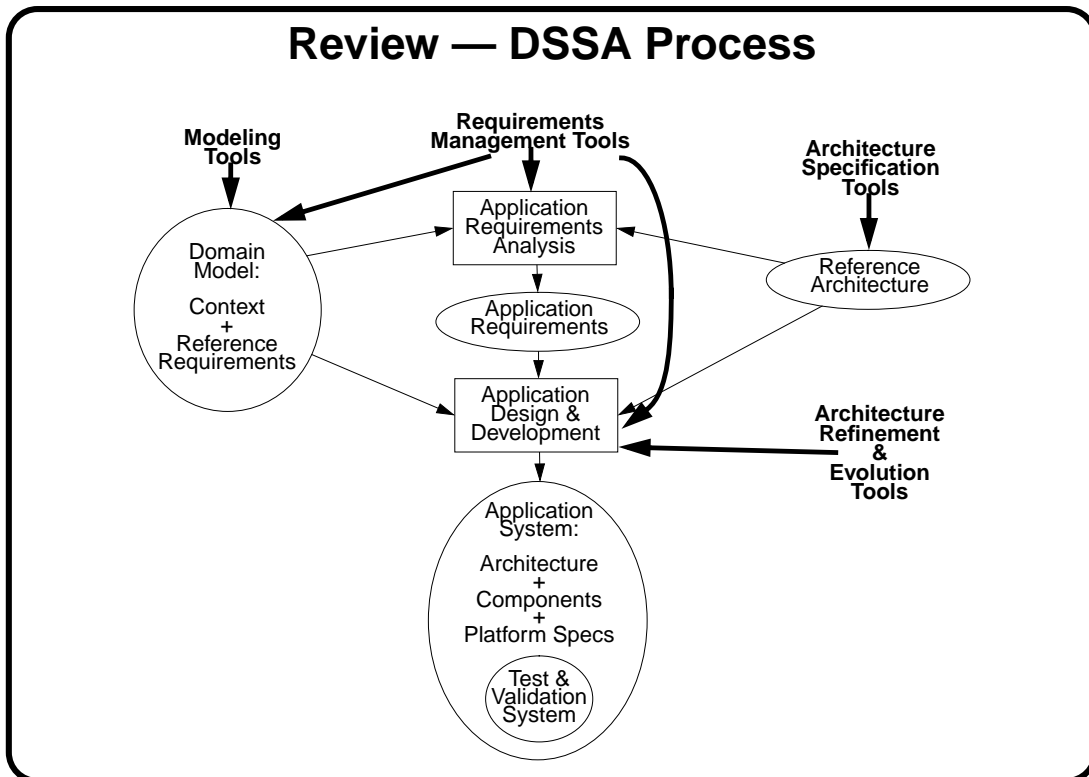


Review — DSSA

- Why domain-specific?
 - development can be optimized
 - reuse potential is maximized
- A DSSA is comprised of
 - a domain model,
 - reference requirements,
 - a reference architecture,
 - its supporting infrastructure/environment, and
 - a process/methodology to instantiate/refine and evaluate it.
- Reference architecture
 - standardized architecture describing all systems in a domain
 - focuses on fundamental domain abstractions

Review — DSSA-Based Software Development





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*

Categories of Styles

- Idioms & patterns
 - deal with global organizational structures
 - application-domain independent
 - to be discussed today
 - pipe and filter
 - client-server
 - blackboard
 - layered
- Reference models
 - specific configurations for certain application areas
 - may be effective outside their initial domains
 - discussed previously
 - canonical compiler architecture
 - other DSSAs

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

Three Views of Architectural Style

	Language	System of Types	Theory
Vocabulary	a set of grammatical productions	a set of types; in OO, sub- and super-types possible;	represented indirectly, in terms of elements' logical properties
Configuration Rules	context-free and -sensitive grammar rules	maintained as type invariants	defined as further axioms
Semantic Interpretations	standard techniques for assigning meaning to languages	operationally realized in the code that modifies type instances	defined as further axioms
Analyses	performed on architectural "programs" (compilation, flow)	dependent on types involved (type checking, code generation)	by proving theorems, thereby extending the theory of the style

Style Analysis Dimensions

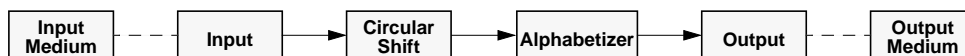
- What is the design vocabulary?
- What are the allowable structural patterns?
- What is the underlying computational model?
- What are the essential invariants of the style?
- What are common examples of its use?
- What are the (dis)advantages of using the style?
- What are the style's specializations?

Some Common Architectural Styles

- “Basic” styles
 - pipe and filter
 - object-oriented
 - implicit invocation
 - layered systems
 - blackboard
 - client-server
 - state transition
- “Derived” styles
 - GenVoca
 - C2

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

Implicit Invocation Style

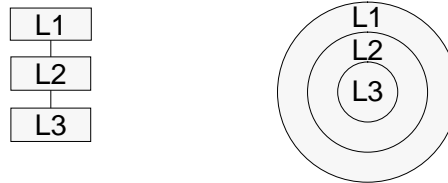
- Event announcement instead of method invocation
 - “listeners” register interest in and associate methods with events
 - the system invokes all registered methods implicitly
- Component interfaces are methods and events
- Two types of connectors
 - invocation is either explicit or implicit in response 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

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
 - service provider to layers “above”
 - service consumer from layers “below”
- Connectors are protocols of layer interaction
- Example — operating systems
- *Virtual machine* style results from fully opaque layers

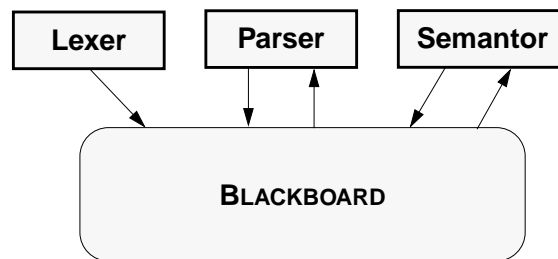


Layered Style (cont.)

- 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

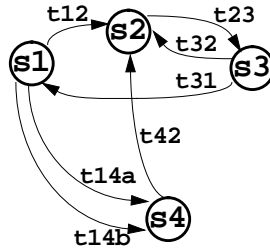


Client-Server Style

- 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

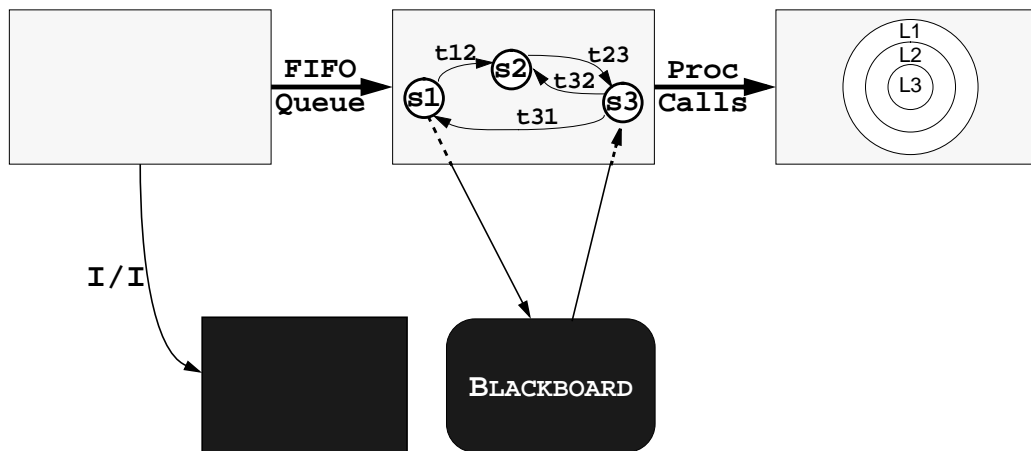
State-Transition Style

- 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

Heterogeneous Styles



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

Open Issues

- [Shaw95] attempts to address some of these
 - use of styles is generally ad-hoc
 - difficult to delimit system aspects that can/should be specified by a style
 - difficult to compare styles based on their properties
 - difficult to relate systems developed in different styles
 - difficult to select appropriate style(s) for a given problem
 - unclear how existing styles can be most effectively combined to produce a new style
 - is a rigorous basis for understanding styles the answer?
- What is the relationship between domains and styles?

