Software Design and Architecture

Lecture 08 & 09

Syed Salman Ahmed

Architectural Patterns

- An architectural pattern is a proven structural organization schema for software systems.
- In a system structured according to the client-server pattern, for instance, two subsystems are distinguished:
  - the client (which can have many instances),
  - and the server (which is unique).
- The responsibility of the client may be to show a user-interface to the user;
- The responsibility of the server may be to process lots of questions, and to guard data that are of interest to the client.

- A pattern also describes rules and guidelines for organizing the relationships among the subsystems.
  - The relationship between client and server is that the client asks questions and the server answers them.
- Patterns are written by people with lots of experience.
- Patterns enable others to learn from experiences of other people.
- They capture existing, well proven solutions in software development, and help to promote good design practices.
- Architectural patterns are also called styles, or standard architectures, but the word architectural style is more often used for a concept less fine-grained than a pattern; several patterns may then belong to the same architectural style.

Why are patterns helpful?

1. **General solution to design problem:** A pattern addresses a recurring design problem, for which a general solution is known among experienced practitioners:
2. **Documents Solution:** A pattern documents existing, well-proved design solutions.
3. **Reuse of Solution:** By writing a pattern, it becomes easier to reuse the solution.
4. **Common Vocabulary:** Patterns provide a common vocabulary and understanding of design solutions.
5. **Design Language:** Pattern names become part of a widespread design language.
6. **No need to explain:** They remove the need to explain a solution to a particular problem with a lengthy description. software architectures.
7. **Original Vision:** They help maintaining the original vision when the architecture is extended and modified, or when the code is modified (but can not guarantee that).
8. **Defined Properties:** Patterns support the construction of software with defined properties.
9. **Building Blocks:** Patterns may be seen as building blocks for a more complicated design.

- When we design a client-server application, for instance, the server should not be built in such a way that it initiates communication with its clients.
  - For example, the MVC (Model-View-Controller) pattern supports changeability of user interfaces.

Pattern schema or template

- **Context:**
  - the situation giving rise to a problem.
- **Problem:**
  - the recurring problem in that context.
  - A solution to the problem should fulfill requirements, consider constraints, and have desirable properties.
  - These conditions are called forces. Forces may conflict with each other (performance may conflict with extensibility, for instance). Forces differ in the degree in which they are negotiable
- **Solution:**
  - a proven solution for the problem.
  - The solution is given as a structure with components and relationships, and as a description of the run-time behavior.
  - The first description is a static model of the solution; the second is a dynamic one.

Design patterns versus architectural patterns

Design Patterns	Architectural Patterns
solution for a common problem in the form of classes working together	Solution in the form of components and connectors
Smaller in scale	Larger in scale
do not influence the fundamental structure of a software system	influence the fundamental structure of a software system
only affect a single subsystem	May affect more than one subsystem
may help to implement an architectural pattern	

For example, the observer pattern (a design pattern) is helpful when implementing a system according to the MVC architectural pattern.

Architectural Patterns versus Architectural Styles

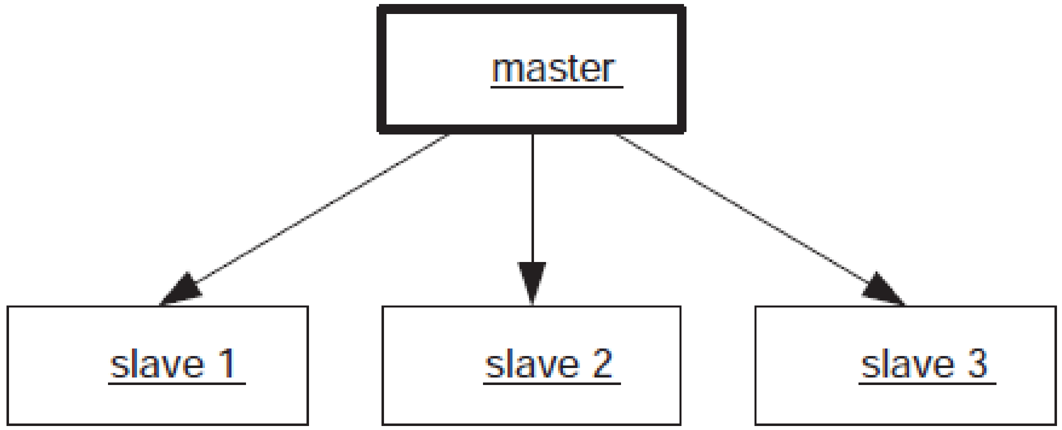
- Patterns have been developed bottom-up: for a given problem, a certain kind of solution has been used over and over again, and this solution has been written down in the form of a pattern.
- Architectural styles on the other hand, have been formulated top-down: when you see a software system as a configuration of components and connectors, you can classify them according to the nature of the components and connectors.
- In general, patterns will belong to one of those styles.
- A style may be:
  - Layered style
  - Data flow style
  - Shared memory style
  - Implicit invocation style

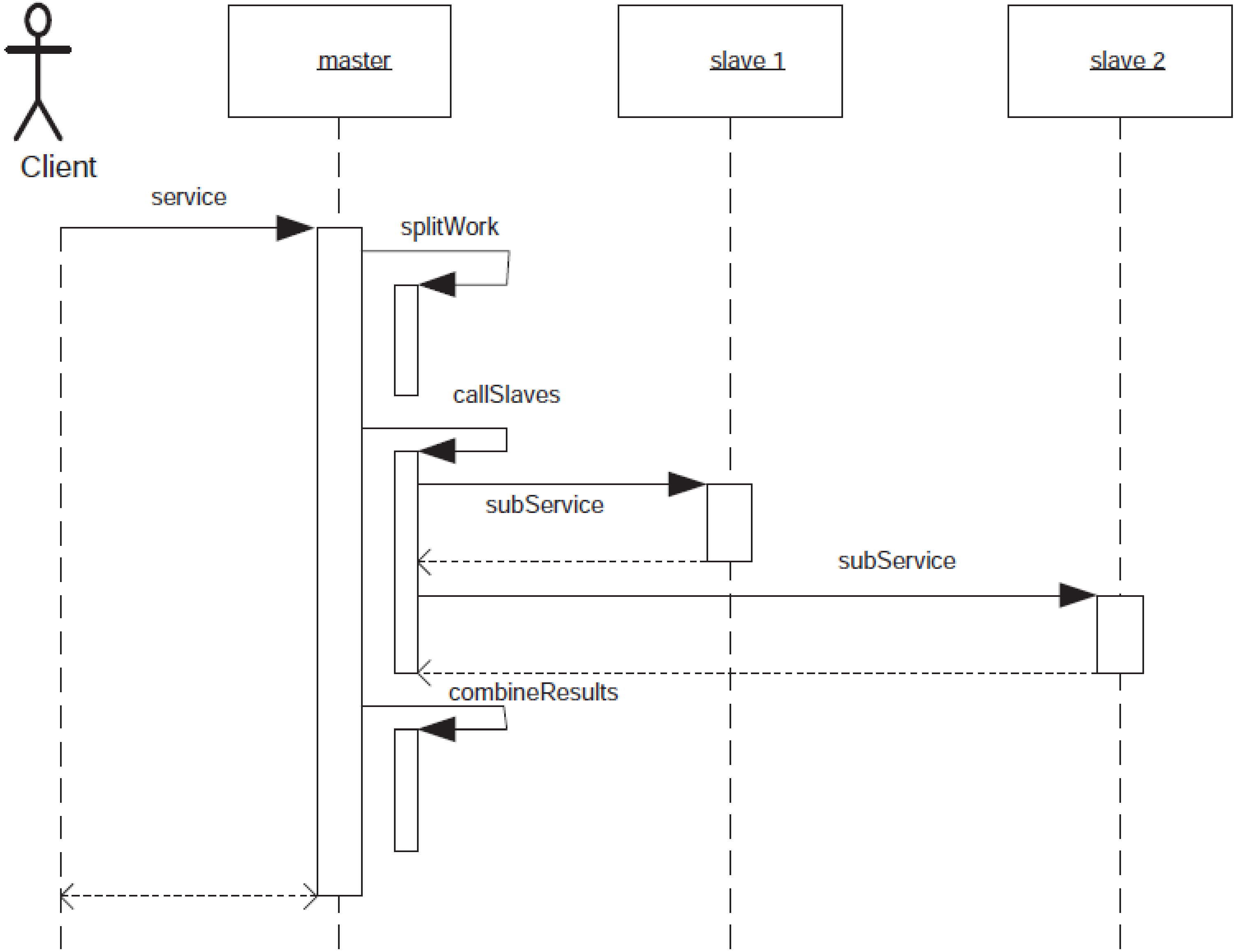
Examples of architectural patterns

- Layered Pattern
- Client-server Pattern
- Pipe filter Pattern
- Event-bus Pattern or Publish-Subscribe Pattern
- Blackboard Pattern
- Interpreter Pattern
- MVC Pattern
- Master Slave Pattern
- Broker Pattern

Master-Slave Pattern

- The Master-slave pattern supports fault tolerance and parallel computation.
- The master component distributes the work among identical slave components, and computes a final result from the results the slaves return.
- The Master-slave pattern is applied for instance in process control, in embedded systems, in large-scale parallel computations, and in fault-tolerant systems.
- Divide and conquer





master
slave 1
slave 2
Client
service
splitWork
callSlaves
subService
subService
combineResults

Examples of Master-Slave

- Fault tolerance support in databases
- Parallel computing
- Computational accuracy (AI)

Issues in Master Slave

- The aspect of coordination is separated from the actual work: concerns are separated.
- The slaves are isolated: there is no shared state.
- They operate in parallel.
- The latency can be an issue.
- The pattern can only be applied to a problem that is decomposable.

Style Summary (1/4)

Style Category & Name	Summary	Use It When	Avoid It When
Language-influenced styles			
Main Program and Subroutines	Main program controls program execution, calling multiple subroutines.	Application is small and simple.	Complex data structures needed. Future modifications likely.
Object-oriented	Objects encapsulate state and accessing functions	Close mapping between external entities and internal objects is sensible. Many complex and interrelated data structures.	Application is distributed in a heterogeneous network. Strong independence between components necessary. High performance required.
Layered			
Virtual Machines	Virtual machine, or a layer, offers services to layers above it	Many applications can be based upon a single, common layer of services. Interface service specification resilient when implementation of a layer must change.	Many levels are required (causes inefficiency). Data structures must be accessed from multiple layers.
Client-server	Clients request service from a server	Centralization of computation and data at a single location (the server) promotes manageability and scalability; end-user processing limited to data entry and presentation.	Centrality presents a single-point-of-failure risk; Network bandwidth limited; Client machine capabilities rival or exceed the server's.

Style Summary, continued (2/4)

Data-flow styles			
Batch sequential	Separate programs executed sequentially, with batched input	Problem easily formulated as a set of sequential, severable steps.	Interactivity or concurrency between components necessary or desirable. Random-access to data required.
Pipe-and-filter	Separate programs, a.k.a. filters, executed, potentially concurrently. Pipes route data streams between filters	[As with batch-sequential] Filters are useful in more than one application. Data structures easily serializable.	Interaction between components required. Exchange of complex data structures between components required.
Shared memory			
Blackboard	Independent programs, access and communicate exclusively through a global repository known as blackboard	All calculation centers on a common, changing data structure; Order of processing dynamically determined and data-driven.	Programs deal with independent parts of the common data. Interface to common data susceptible to change. When interactions between the independent programs require complex regulation.
Rule-based	Use facts or rules entered into the knowledge base to resolve a query	Problem data and queries expressible as simple rules over which inference may be performed.	Number of rules is large. Interaction between rules present. High-performance required.

Style Summary, continued (3/4)

Interpreter			
Interpreter	Interpreter parses and executes the input stream, updating the state maintained by the interpreter	Highly dynamic behavior required. High degree of end-user customizability.	High performance required.
Mobile Code	Code is mobile, that is, it is executed in a remote host	When it is more efficient to move processing to a data set than the data set to processing. When it is desirous to dynamically customize a local processing node through inclusion of external code	Security of mobile code cannot be assured, or sandboxed. When tight control of versions of deployed software is required.

Style Summary, continued (4/4)

Implicit Invocation			
Publish-subscribe	Publishers broadcast messages to subscribers	Components are very loosely coupled. Subscription data is small and efficiently transported.	When middleware to support high-volume data is unavailable.
Event-based	Independent components asynchronously emit and receive events communicated over event buses	Components are concurrent and independent. Components heterogeneous and network-distributed.	Guarantees on real-time processing of events is required.
Peer-to-peer	Peers hold state and behavior and can act as both clients and servers	Peers are distributed in a network, can be heterogeneous, and mutually independent. Robust in face of independent failures. Highly scalable.	Trustworthiness of independent peers cannot be assured or managed. Resource discovery inefficient without designated nodes.
More complex styles			
C2	Layered network of concurrent components communicating by events	When independence from substrate technologies required. Heterogeneous applications. When support for product-lines desired.	When high-performance across many layers required. When multiple threads are inefficient.

Style Summary (1/4)

Style Category & Name	Summary	Use It When	Avoid It When
Language-influenced styles			
Main Program and Subroutines	Main program controls program execution, calling multiple subroutines.	Application is small and simple.	Complex data structures needed. Future modifications likely.
Object-oriented	Objects encapsulate state and accessing functions	Close mapping between external entities and internal objects is sensible. Many complex and interrelated data structures.	Application is distributed in a heterogeneous network. Strong independence between components necessary. High performance required.
Layered			
Virtual Machines	Virtual machine, or a layer, offers services to layers above it	Many applications can be based upon a single, common layer of services. Interface service specification resilient when implementation of a layer must change.	Many levels are required (causes inefficiency). Data structures must be accessed from multiple layers.
Client-server	Clients request service from a server	Centralization of computation and data at a single location (the server) promotes manageability and scalability; end-user processing limited to data entry and presentation.	Centrality presents a single-point-of-failure risk; Network bandwidth limited; Client machine capabilities rival or exceed the server's.

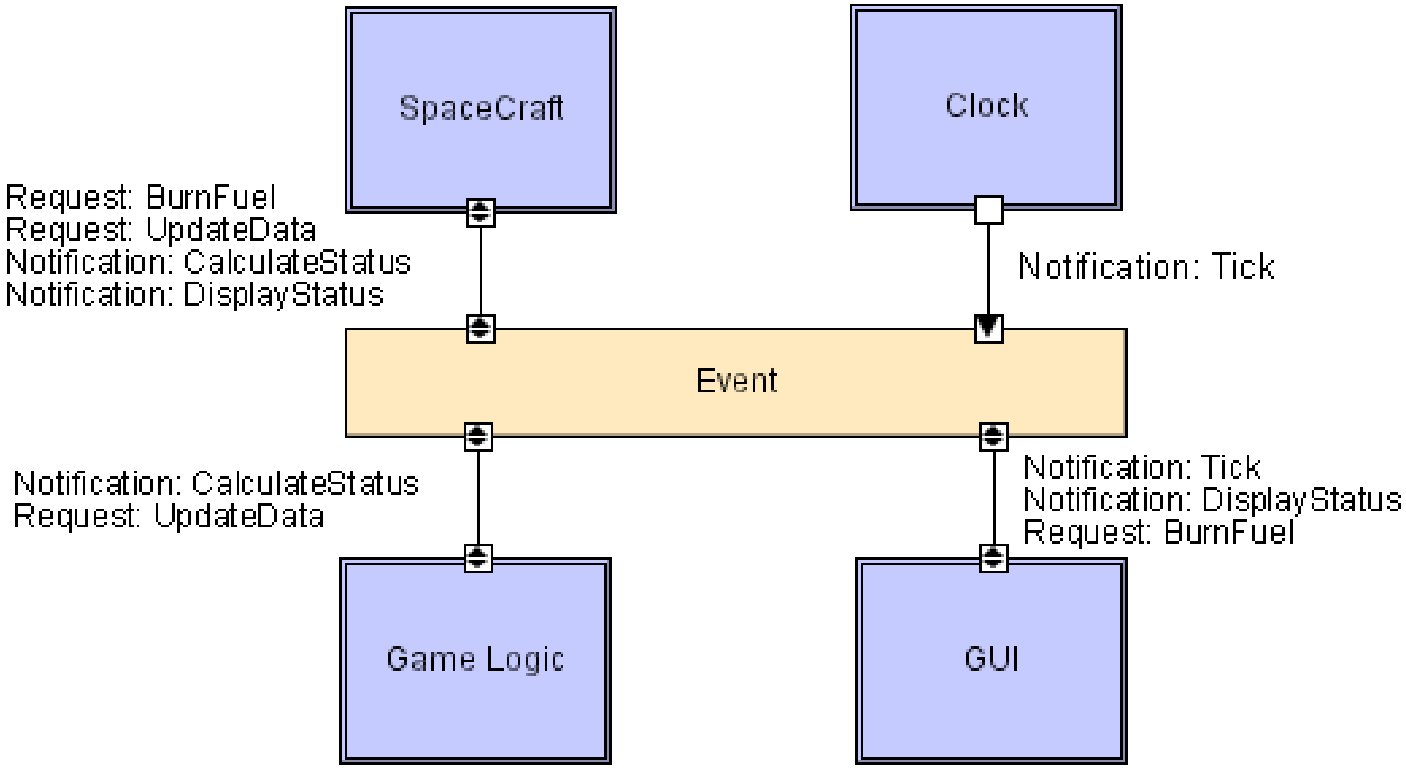
C2 Style

An indirect invocation style in which independent components communicate exclusively through message routing connectors. Strict rules on connections between components and connectors induce layering.

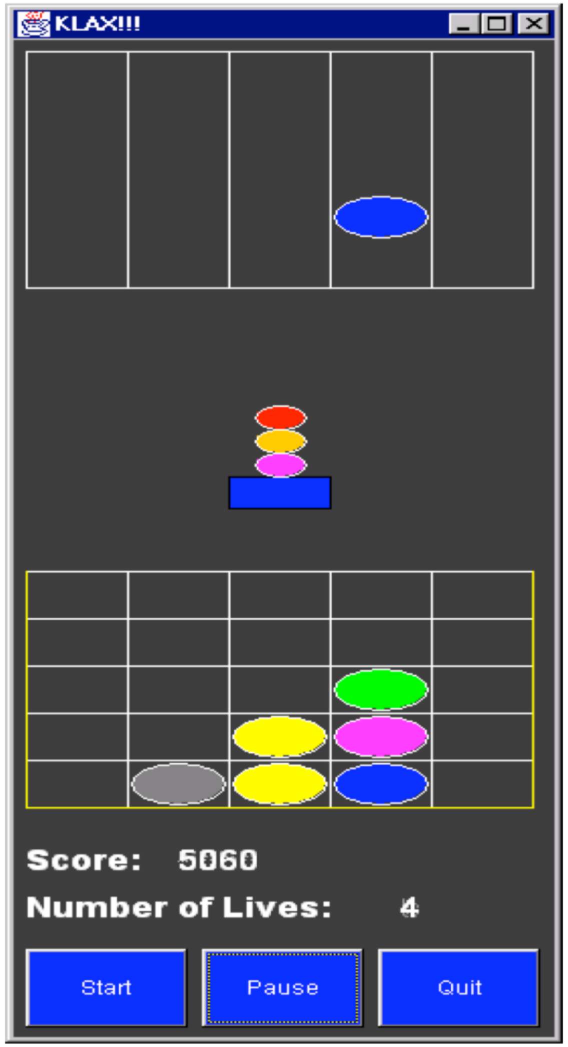
C2 Style (cont'd)

- Components: Independent, potentially concurrent message generators and/or consumers
- Connectors: Message routers that may filter, translate, and broadcast messages of two kinds: notifications and requests.
- Data Elements: Messages – data sent as first-class entities over the connectors. Notification messages announce changes of state. Request messages request performance of an action.
- Topology: Layers of components and connectors, with a defined "top" and "bottom", wherein notifications flow downwards and requests upwards.

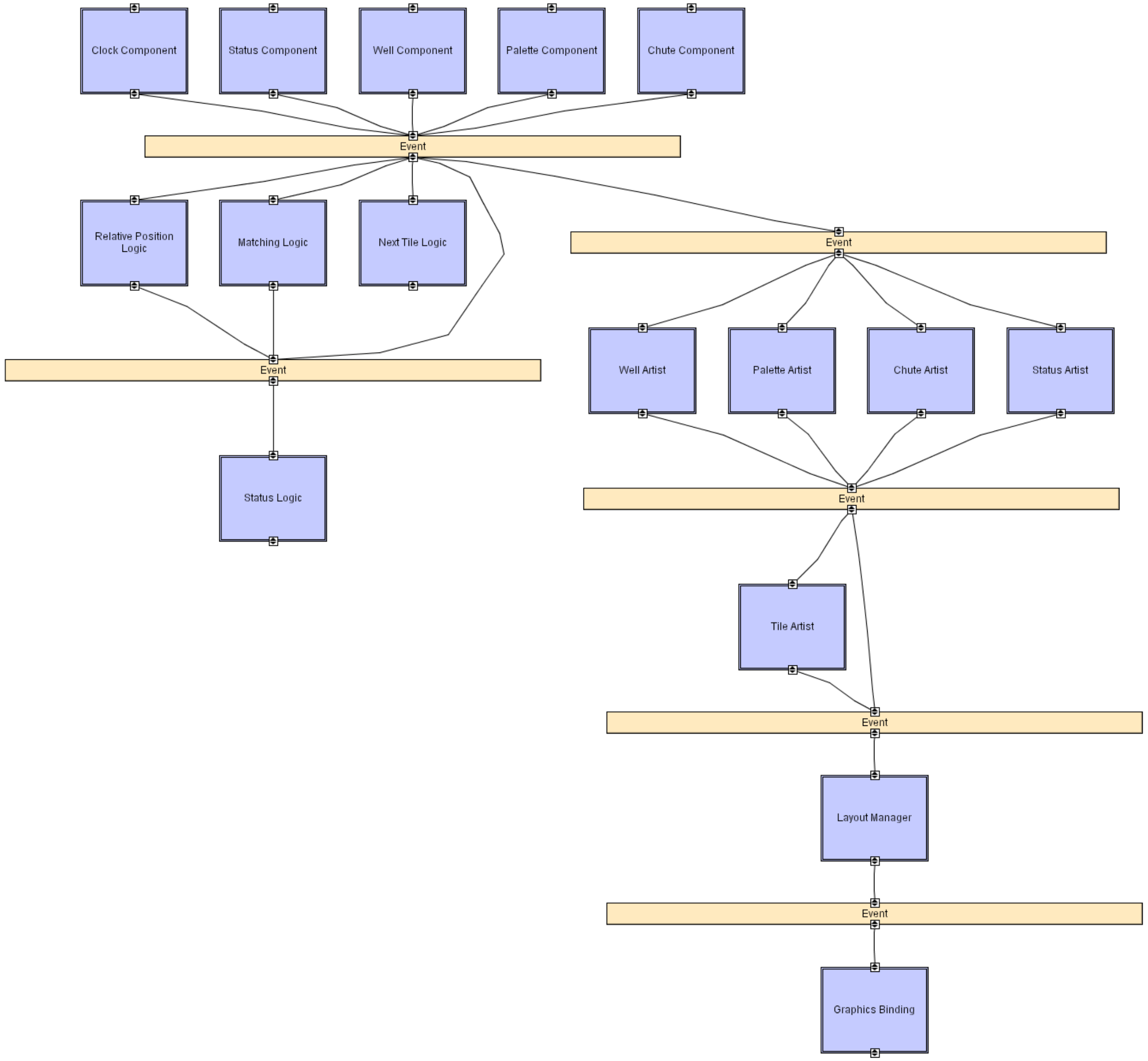
C2


SpaceCraft
Clock
Request: BurnFuel
Request: UpdateData
Notification: CalculateStatus
Notification: DisplayStatus
Notification: Tick
Event
Notification: CalculateStatus
Request: UpdateData
Notification: Tick
Notification: DisplayStatus
Request: BurnFuel
Game Logic
GUI

KLAX



KLAX in C2



Clock Component
Status Component
Well Component
Palette Component
Chute Component
Event
Relative Position Logic
Matching Logic
Next Tile Logic
Event
Event
Status Logic
Well Artist
Palette Artist
Chute Artist
Status Artist
Event
Tile Artist
Event
Layout Manager
Event
Graphics Binding

Observations

- Different styles result in
  - Different architectures
  - Architectures with greatly differing properties
- A style does not fully determine resulting architecture
  - A single style can result in different architectures
  - Considerable room for
    - Individual judgment
    - Variations among architects
- A style defines domain of discourse
  - About problem (domain)
  - About resulting system

Design Recovery

- What happens if a system is already implemented but has no recorded architecture?
- The task of design recovery is
  - examining the existing code base
  - determining what the system's components, connectors, and overall topology are.
- A common approach to architectural recovery is clustering of the implementation-level entities into architectural elements.
  - Syntactic clustering
  - Semantic clustering

Syntactic Clustering

- Focuses exclusively on the static relationships among code-level entities
- Can be performed without executing the system
- Embodies inter-component (a.k.a. coupling) and intra-component (a.k.a. cohesion) connectivity
- May ignore or misinterpret many subtle relationships, because dynamic information is missing

Semantic Clustering

- Includes all aspects of a system's domain knowledge and information about the behavioral similarity of its entities.
- Requires interpreting the system entities' meaning, and possibly executing the system on a representative set of inputs.
- Difficult to automate
- May also be difficult to avail oneself of it

When There’s No Experience to Go On

- The first effort a designer should make in addressing a novel design challenge is to attempt to determine that it is genuinely a novel problem.
- Basic Strategy
  - Divergence – shake off inadequate prior approaches and discover or admit a variety of new ideas
  - Transformation – combination of analysis and selection
  - Convergence – selecting and further refining ideas
- Repeatedly cycling through the basic steps until a feasible solution emerges.

Analogy Searching

- Examine other fields and disciplines unrelated to the target problem for approaches and ideas that are analogous to the problem.
- Formulate a solution strategy based upon that analogy.
- A common “unrelated domain” that has yielded a variety of solutions is nature, especially the biological sciences.
    - E.g., Neural Networks

Brainstorming

- Technique of rapidly generating a wide set of ideas and thoughts pertaining to a design problem
  - without (initially) devoting effort to assessing the feasibility.
- Brainstorming can be done by an individual or, more commonly, by a group.
- Problem: A brainstorming session can generate a large number of ideas… all of which might be low-quality.
- The chief value of brainstorming is in identifying categories of possible designs, not any specific design solution suggested during a session.
- After brainstorm the design process may proceed to the Transformation and Convergence steps.

“Literature” Searching

- Examining published information to identify material that can be used to guide or inspire designers
- Many historically useful ways of searching “literature” are available
- Digital library collections make searching extraordinarily faster and more effective
  - IEEE Xplore
  - ACM Digital Library
  - Google Scholar
- The availability of free and open-source software adds special value to this technique.

Morphological Charts

- The essential idea:
  - identify all the primary functions to be performed by the desired system
  - for each function identify a means of performing that function
  - attempt to choose one means for each function such that the collection of means performs all the required functions in a compatible manner.
- The technique does not demand that the functions be shown to be independent when starting out.
- Sub-solutions to a given problem do not need to be compatible with all the sub-solutions to other functions in the beginning.

Removing Mental Blocks

- If you can’t solve the problem, change the problem to one you can solve.
  - If the new problem is “close enough” to what is needed, then closure is reached.
  - If it is not close enough, the solution to the revised problem may suggest new venues for attacking the original.

Controlling the Design Strategy

- The potentially chaotic nature of exploring diverse approaches to the problem demands that some care be used in managing the activity
- Identify and review critical decisions
- Relate the costs of research and design to the penalty for taking wrong decisions
- Insulate uncertain decisions
- Continually re-evaluate system “requirements” in light of what the design exploration yields

Insights from Requirements

- In many cases new architectures can be created based upon experience with and improvement to pre-existing architectures.
- Requirements can use a vocabulary of known architectural choices and therefore reflect experience.
- The interaction between past design and new requirements means that many critical decisions for a new design can be identified or made as a requirement

Insights from Implementation

- Constraints on the implementation activity may help shape the design.
- Externally motivated constraints might dictate
  - Use of a middleware
  - Use of a particular programming language
  - Software reuse
- Design and implementation may proceed cooperatively and contemporaneously
  - Initial partial implementation activities may yield critical performance or feasibility information

Heterogeneous Styles

- More complex styles created through composition of simpler styles
- REST (from the first lecture)
  - Complex history presented later in course
- C2
  - Implicit invocation + Layering + other constraints
- Distributed objects
  - OO + client-server network style
  - CORBA

References

- *Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; Chapter 4