Meta-Architecture Document

Lori Gildersleeve

Jennifer Martin  
Cody Nelson

**Table of Contents**

[Architectural Vision 3](#_Toc385427413)

[Principles 3](#_Toc385427414)

[Styles 4](#_Toc385427415)

Style [Object-Oriented] 4

Style [Blackboard] 5

Style [Pipe-and-Filter] 6

[Patterns & Interconnection Mechanisms 7](#_Toc385427416)

[Pattern [State-Logic-Display (Three-Tier)] 7](#_Toc385427417)

[Interconnection Mechanisms 8](#_Toc385427418)

[Philosophies & Preferences 9](#_Toc385427420)

[Guidelines & policies 9](#_Toc385427421)

[Additional Information 9](#_Toc385427422)

Meta-Architecture Document

[Gildersleeve, Martin, Nelson]

# Architectural Vision

The architecture will be the guide by which the programmers will use to develop the Sudoku game. The vision is to create a concise architectural design that

1. clearly separates and defines each process of the Sudoku application,
2. allows developers to follow design easily,
3. allows for future scalability development, and
4. provides for future adaptation.

# Principles

|  |  |
| --- | --- |
| Principle Name | **Separation of Concerns** |
| Description | Creation of independent parts by subdivision of a problem |
| Rationale/Benefits | Aides in encapsulation of functionality and defining classes |
| Implications | User interface, business rules, and data access can be kept independent of one another |
| Counterargument | Parts of the problem could not be kept independent without significant trade-offs of performance, functionality or appearance. |

|  |  |
| --- | --- |
| Principle Name | **Scalable Components** |
| Description | Create each component with a single, clearly defined purpose using simple, understandable interfaces that encompass only functional responsibilities |
| Rationale/Benefits | Keeps classes simple and well-defined |
| Implications | Components can encapsulate ideas with relative ease |
| Counterargument | Components cannot be kept independent without significant trade-offs of performance, functionality, or appearance. |

# Styles

### **Style [Object-Oriented]**

### **Summary Description**: States and functions encapsulated by objects

### **Components**: Objects (AKA instances of a class)

### **Connectors**: Method invocation (procedure calls to manipulate state)

### **Data Elements**: Arguments to methods

### **Topology**: Can vary arbitrarily; components may share data and interface functions through inheritance hierarchies

### **Additional** **Constraints**: Commonly shared memory, single threaded

### **Qualities** **Yielded**: Integrity of data operations; data manipulated only by appropriate functions; abstraction implementation details hidden

### **Typical** **Uses**: Applications where the designer wants a close correlation between entities in the physical world and entities in the program; pedagogy; applications involving compiles, dynamic data structures

### **Cautions**: Use in distributed applications requires extensive middleware to provide access to remote objects; relatively inefficient for high-performance applications with large, regular numeric data structures; lack of additional structuring principles can result in highly complex applications.

### **Relation** **to** **Programming** **Languages**: Java, C++

**Rationale**: Allows the use of Java, which is familiar to all team members; application idea (Sudoku) can be encapsulated simply via object-oriented techniques (classes).

**Style [Blackboard]**

### **Summary Description**: Global data repository (blackboard) through which independent programs communicate

### **Components**: Blackboard and independent programs (AKA knowledge sources)

### **Connectors**: Database query, method call, or direct memory reference can access the blackboard

### **Data Elements**: Blackboard stored data

### **Topology**: Blackboard is at the center of this Star Topology

### **Additional** **Constraints**: Programs can ask blackboard if there is new data OR blackboard can push notifications of new data/updates

### **Qualities** **Yielded**: Complex problems do not have to have preplanned solutions

### **Typical** **Uses**: Artificial Intelligence experimental solutions to problems; shared memory applications

### **Cautions**: Not ideal if: a well-defined solution to a problem exists, complex regulations are needed between the independent programs, or data representation is subject to frequent change.

### **Relation** **to** **Programming** **Languages**: Synchronization and concurrency primitives are required for versions that allow concurrency

**Rationale for Not Using in our Solution**: Sudoku requires constant updates and communication to the data layer. Also, this is a well-defined gamed at this juncture, and therefore, well-defined solutions are attainable.

**Style [Pipe-and-Filter]**

### **Summary Description**: Streams used to pass data from one program to another; separate programs executed independently with option of concurrency

### **Components**: Filters (AKA independent programs)

### **Connectors**: Data router streams; operating system service

### **Data Elements**: Linear data stream; not explicit

### **Topology**: Pipeline

### **Additional** **Constraints**:

### **Qualities** **Yielded**: Simplicity of data streams (in and out) facilitates creation of unique combinations of filters for new programs; filters are autonomous

### **Typical** **Uses**: Universal in operating system programming

### **Cautions**: Not ideal if: programs need to interact or exchange of complex data structures is needed

### **Relation** **to** **Programming** **Languages**: Unix shells

**Rationale for Not Using in our Solution**: If the Sudoku application was being built for command line prompts, this style could be useful. Since the application will be web based, this style is not ideal. The data passed between the classes is very complex and communication between classes is ideal.

# Patterns & interconnection Mechanisms

Three types of patterns are of interest:

* Structural Patterns
* Interconnection Patterns
* Decoupling Patterns

These follow directly from describing architecture in terms of a set of components and their relationships.

Patterns are described in terms of a Pattern Form (see below), and one or more CRC “cards” (see below), which describe each of the major pieces of the pattern. Cut and paste copies of the Pattern Form and CRC table (as needed), for each Pattern or Interconnect Mechanism described. You may not think of every pattern the first tie through. You may struggle with rationale. That’s OK. Plan time in your project to revisit this document.

If you are thinking of following a reference architecture, this is the place to describe and consider it.

## Pattern [State-Logic-Display (Three-Tier)]

### **Summary Description**: Three-tiered architecture where there is a data source behind set of logic rules, and the logic is accessed by a user interface.

### **Context of Use (Intent):** To have a maintainable, simple architecture that keeps presentation, application processing, and data management functions separate.

### **Problem Statement**: How will we use the three-tier pattern to formulate our program’s architecture?

### **Solution Description**: Carefully construct each tier by determining its responsibilities and functionality.

### **Variants and Related Patterns**: Model-View-Controller

### **Known Uses**: Business applications, multi-player games, and web-based applications

### **Consequences**: Segregation of an application into tiers allows us to modify tiers individually without rework of the entire program.

**Rationale:** The three-tier pattern is a simple, flexible architecture that is maintainable and fits our project needs.

## Interconnection Mechanism

## [Procedure Call Connector]

**Summary Description**: Model the flow of control among components through various invocation techniques, and perform transfer of data among the interacting components through the use of parameters and return values.

**Rationale:** Since the Sudoku application is a synchronous blocking process and method calls have a single source and destination, this connector type will be most useful.

## [Remote Procedure Call Connector]

**Summary Description**: Model the flow of control among components through various invocation techniques, and perform transfer of data among the interacting components through the use of parameters and return values. These connectors communicate via a framework that allows calls to occur across a network connection while still appearing to be within the same virtual machine.

**Rationale:** This type of connector would be useful if the Sudoku application was deployed on multiple machines. Then calls from one machine, containing the GUI, would be passed across a network connection to a second machine, containing the data model and update processing, and back again. This connector is good to keep in mind for scalability needs in the future. Should the application evolve into a multi-user/platform application, this connector type can be easily constructed to handle the additional communications.

**[Event Connector]**

**Summary Description:** Invoked by an event and passes messages/control to interested parties.

**Rationale:** This type of connector could be used in the UI sector of our application. For instance, a mouse click in a cell of the Sudoku puzzle would be the invoking event to send a message to the business logic section of the application to pop up an input box for the user to input their guess for the value of the cell.

**[Data Access Connector]**

**Summary Description:** Allows component access to data store

**Rationale:** Sudoku application has a mock database component housing puzzle values, guesses, locations, etc. Data Access connectors will allow the business logic tier of the application to gain and manipulate data housed in this component. For instance, a user inputs a guess into a cell on the Sudoku puzzle. The business logic tier will validate whether it’s a valid input. If it is, a data access connector will be invoked to update the database guess value for the cell. This type of connector will allow for scalability and adaptability in the future. For instance, if the database or data in general was refactored, this connector would insulate the user from any knowledge of the change. The changed data could be massaged for appropriate input before being sent back to the caller or the necessity for additional code changes to the application should a refactor occur.

# Philosophies & Preferences

1. Construct a concisely detailed architecture for our project to solve design decisions.
2. Document architecture development thoroughly by creating these documents:
   1. Conceptual Architecture
   2. Interface Control
   3. Logical Architecture
   4. Execution Architecture
3. Complete all documents, processes and development required to submit our Sudoku application and receive a grade of A+.

# Guidelines & policies

1. Reach goals throughout the semester via individual and team weekly goals based on deliverables due dates.
2. Utilize our professor’s knowledge and skill as needed.

# Additional Information

1. Our project will be stored and accessible to each team member via a Git repository.   
   Access the repository at: **https://github.com/scodynelson/Sudoku.git**
2. Our project will use the following languages and tools:
   1. **Java SDK 7** – main programming language  
      (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html)
   2. **Groovy 2.2.2** – secondary programming language  
      (http://groovy.codehaus.org/Download)
   3. **Maven 3.2.1** – build manager to handle outside dependencies  
      (http://maven.apache.org/download.cgi)
   4. **IntelliJ IDEA 13 Community Edition** – integrated development environment  
      (http://www.jetbrains.com/idea/download/)
   5. **GIT** – version control  
      (http://git-scm.com/)