



## Kanji Architectural Pattern

---

Kazuki Kimura

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

March 9, 2023

# Kanji Architectural Pattern

Kazuki Kimura

*Transport and Telecommunication institute  
Riga, Latvia, Lomonosova street 1,  
e-mail [kazuki.kimura.jp@gmail.com](mailto:kazuki.kimura.jp@gmail.com)*

**Keywords:** - services, pattern, development, framework, structure, solid, components.

## **Abstract:**

The actuality of this research lies in its presentation of the Kanji architectural pattern as a new approach to web application development. This pattern offers greater flexibility and adaptability for designing and implementing software systems with a strong emphasis on reusability and modularity. This is particularly important in the modern context of software development, where the need for responsive and engaging user interfaces is crucial and applications can span thousands of files and lines of code.

Research is devoted to problematizing the development of modern architectural patterns, such as Model-View-Controller (MVC), Model-View-Presenter (MVP), Model-View-ViewModel (MVVM), Atomic, and others, see section 1-3. These patterns were developed to solve software design issues but have limitations that affect the development process. One of the main problems with these patterns is that they tend to create tightly coupled code, making it difficult to change or maintain. Additionally, these patterns were developed during a time when applications did not have access to modern tools and development approaches, leading to longer development times and increased complexity. Moreover, architectural patterns often do not address cross-cutting concerns, such as component samples and component reusability. This research aims to provide a comprehensive analysis of these issues and offer potential solutions to improve the development process.

The goal of the research is to analyze existing architectural pattern approaches and determine how the newly developed Kanji architectural pattern can resolve contemporary issues to provide a comprehensive analysis and offer potential solutions to improve the development process.

The Kanji architectural pattern consists of five main components: Bookmark, Reference, Remark, Paper and Page, each of which is designed to be modular, self-contained, and highly reusable. And submodular components such as: Stroke, Radical, Word and Paragraph, see section 5-6. That submodular components gives possibility to create application view structure with reusability principles.

Author provides a detailed description of each component, as well as guidelines and best practices for their use in software design and implementation, based on S.O.L.I.D that is a mnemonic acronym for five design principles intended to make object-oriented designs more understandable, flexible, and maintainable (Erinç Kemal Yiğit, 2020). The performance and

maintainability of software systems designed using the Kanji pattern are evaluated in comparison to other architectural patterns, demonstrating its effectiveness in building responsive and engaging user interfaces.

Furthermore, the suitability of the Kanji pattern for different types of software systems and domains is investigated, showing its potential to be used in a wide range of applications, including those in the context of the emerging Society 5.0 (BrandNewsPicks, 2021). That one of the main aims for Society 5.0 is to reuse components of Kanji architectural pattern at any applications without additional time spending on maintenance and restructuration.

Overall, this research presents a new approach to web application development that combines the strengths of existing architectural patterns in a flexible and adaptable way, providing a solid foundation for building sophisticated and responsive software systems.

*The research is supervised  
by Dr.sc.ing., Professor Irina Yatskiv*

## **Introduction**

Software architecture patterns have emerged as a valuable tool for software engineers to design, communicate, and evaluate software systems. In particular, the Kanji architectural pattern has been proposed as a novel approach to designing software systems that are flexible, adaptable, and scalable. The Kanji pattern is based on the idea of using a set of building blocks or "Kanjis" that can be combined in different ways to form software components.

The aim of this work is to investigate the feasibility and potential of the Kanji architectural pattern as a means of improving the quality and performance of software development processes. Reuse components of kanji architectural pattern at any applications without additional time spending on maintenance and restructuration that gives possibility to use pattern for Society 5.0.

In particular, the work focuses on the practical implementation of the Kanji pattern in real-world software projects, with the goal of demonstrating its effectiveness and identifying its strengths and weaknesses.

To achieve this aim, the following objectives are pursued:

- To develop a set of guidelines and best practices for using the Kanji pattern in software design and implementation.
- To evaluate the performance and maintainability of software systems designed using the Kanji pattern in comparison to other architectural patterns.
- To investigate the suitability of the Kanji pattern for different types of software systems and domains.

- Creating independent library modules

The central research hypothesis is that the Kanji architectural pattern can provide a flexible, adaptable, and scalable approach to software design that can improve the quality and performance of software systems. Specifically, it is hypothesized that the use of Kanjis can facilitate the design of software components that are modular, reusable, and composable, thereby reducing the complexity and improving the maintainability of the software system.

The object of the research is the Kanji architectural pattern, with a focus on its practical implementation in software projects. The subject of the research is the investigation of the effectiveness and suitability of the Kanji pattern for software systems, with an emphasis on its performance, maintainability, and scalability.

The methods used in this work include the development of a set of guidelines and best practices for using the Kanji pattern, the implementation of software systems using the Kanji pattern, and the evaluation of the performance and maintainability of the software systems using various metrics and techniques.

The expected result of this work is the identification of the strengths and weaknesses of the Kanji architectural pattern, as well as the development of a set of guidelines and best practices for using the pattern in software design and implementation. Additionally, the work is expected to demonstrate the effectiveness and suitability of the Kanji pattern for different types of software systems and domains.

In summary, this work aims to investigate the potential and feasibility of the Kanji architectural pattern as a novel approach to software design. By evaluating its performance and suitability in real-world software projects, this work aims to contribute to the development of best practices and guidelines for using the Kanji pattern, and to promote its adoption as a valuable tool for software engineers.

## **1. What is kanji?**

Hieroglyphic writing has existed for thousands of years and has been used by many ancient civilizations. The concepts behind this type of writing, which involve the use of symbols to represent words and ideas, are still relevant today. In fact, it can be argued that hieroglyphic writing is an important precursor to modern-day programming languages.

Despite the fact that programming languages are commonly used today, the syntax and semantics of non-programming languages are like those used in programming. Both types of languages have their own set of rules and guidelines for usage, as well as specific writing styles. Writing styles can take many forms, such as hieroglyphic, Cyrillic, Latin, and more.

This thesis will focus primarily on the hieroglyphic style of writing, which offers the unique ability to quickly and easily understand the meanings conveyed in words, sentences, and phrases. Using the principles of hieroglyphic modular structure, this thesis will explore how to build a front-end architecture with a modular structure on an architectural level, making it easier to memorize dependencies. The concepts presented in this thesis are not limited to front-end architecture and can be applied to back-end architecture as well. This thesis will provide an overview of the principles involved in building a modular architecture and provide practical examples and tips for implementing these concepts.

By the end of this thesis, readers will have a solid understanding of the principles involved in building a modular architecture and will be able to apply this knowledge to their own projects. It is important to note that while this thesis focuses primarily on front-end architectures, the concepts presented can be applied to back-end architectures as well.

### **1.1. Kanji structure**

This section provides an overview of the structure of hieroglyphics, explaining how each section of kanji is constructed and how they are interconnected. The aim of this chapter is to prepare the reader to understand the basic definitions and their connections, especially if they are not familiar with the definitions of semantics, syntax, hieroglyphs, and radicals.

It's important to note that the concept of hieroglyphs exists in many languages, but for the purpose of this work, the definition of kanji is used from Japanese and translated verbatim as “han” characters, or in other words, Chinese characters. These characters have a complex structure, which is classified in dictionaries according to their main modules, known as strokes and radicals (roots). These parts are the smallest components and can have their own definition or up to several meanings.

The following is an example of the complete structure of one of the hieroglyphics.

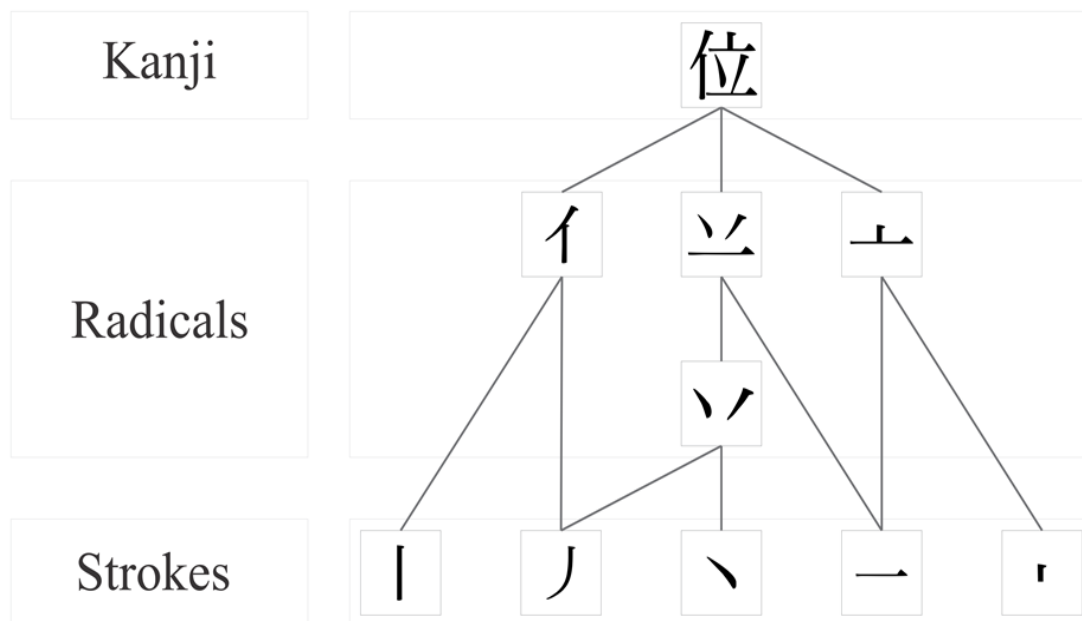


Figure 1.1. Kanji structure

The construction of kanji is based on the following levels of hieroglyphics: from the strokes, you can build more complex parts of the drawing, and from these parts, you can build radicals. From the constructed radicals, you can build more complex radicals or kanji themselves. Moreover, from the kanji themselves, you can build more complex structures of hieroglyphics.








To better understand the structure of hieroglyphics, the following chapter sections will provide detailed descriptions of each of the hieroglyphic components, including strokes, radicals, and kanji, along with their brief meanings. By studying these components, readers will be able to appreciate the intricate beauty of hieroglyphics and understand the power of using them in both front-end and back-end architectures. The knowledge gained from this chapter will also be useful for anyone interested in learning more about the Japanese or Chinese languages and cultures.

### 1.1.1. Strokes

At its most fundamental level, kanji is composed of strokes. Each stroke contains the necessary movements to form a portion of a kanji character. While these components do not carry individual meanings, they convey a sense of feeling or instinctive behavior. Additionally, they lack a fixed position within any radical or hieroglyphic.

The "han" stroke order system was developed to produce aesthetically pleasing and balanced characters on paper, while minimizing hand movement and maximizing stroke efficiency. This book aims to implement the stroke component across the page with minimal coding while maintaining symmetry and balance.

Below are examples of strokes drawing style and their definition:

Stroke	Definition
	Diagonal sweeping
	Horizontal
	Vertical
	Vertical stroke with a hook
	Press down
	Dot
	Vertical stroke with a hook

*1. table. Strokes example*

To understand the principles behind kanji, a detailed explanation of "The Eight Principles of Yong" is not necessary for the purposes of this study. This principle explains how to write common strokes in regular script, which can be found in a single character. While we will not delve into the details of this principle, the diagram provides an illustration of the process, which can help to establish a connection between object-oriented programming and the concept of kanji.

When it comes to the stroke order system, there are several key principles to consider. Firstly, each stroke should be written in a way that produces the most aesthetical and balanced characters on paper. Secondly, strokes should be written in a way that is efficient, minimizing hand

movement and maximizing stroke efficiency. Finally, strokes should be written in a way that maintains symmetry and balance across the page.

However, examining the diagram that illustrates the principle can provide valuable insights into the relationship between object-oriented programming and the concept of kanji:



Figure 1.2. The Eight Principles of Yong








As the diagram of "The Eight Principles of Yong" demonstrates, the principle itself and the combination processes of object-oriented components share a similar idea of creating more complex objects. This connection presents a unique opportunity to explore new paths of relations between the common hieroglyphics structure and the core principles of object-oriented programming.

Through the application of object-oriented programming concepts, it is possible to develop a new perspective based on kanji structure. By leveraging the shared principles of these two seemingly disparate fields, it may be possible to uncover new insights into the structure and meaning of kanji characters.









### 1.1.2. Radicals

Another important concept in kanji is the idea of radicals, which are the building blocks of kanji characters. Radicals have their own meanings, and understanding the meaning of each radical can help to understand the meaning of the kanji character as a whole. The position of the radical in the character can also have an impact on its meaning. For example, a radical placed at the top of a character can indicate heaven or sky, while a radical placed at the bottom can indicate the ground or earth.:

<b>Position</b>	<b>Name</b>	<b>Definition</b>
	<b>Kamae</b>	Surround / left and bottom element
	<b>Ashi</b>	Foot element
	<b>Tsukuri</b>	Right accompanying element
	<b>Kanmuri</b>	Crown element
	<b>Hen</b>	Left sided element
	<b>Tare</b>	“hang down”
	<b>Nyoo</b>	wrap around the bottom of a character

*2. table. Radical positions list*

We will not consider all the existing options with all positions, but to understand how radicals look like depending on the category, below will be provided some examples of radicals drawing style and their meanings:

Radical	Position	Meaning
匸		Hide
儿		Human legs
刂		Knife
冫		Lid, top
亻		Person
勹		To wrap

*3. table. Radical list example*

Radicals are the building blocks of kanji characters, and they can be categorized into two groups:

- Simplified
- Complicated

Simplified radicals can only be used as an add-on element, while complicated radicals can be used as a separate hieroglyphic with additional possible meanings/definitions or as an add-on element for other hieroglyphic parts.

To better understand these categories, we can draw a comparison to web components, such as the "Article" component. A simplified implementation of this component would involve using it as a part of another component, with static size and padding. An example of a simplified web component can be seen in Picture 1-3:

### Thumbnail Gallery

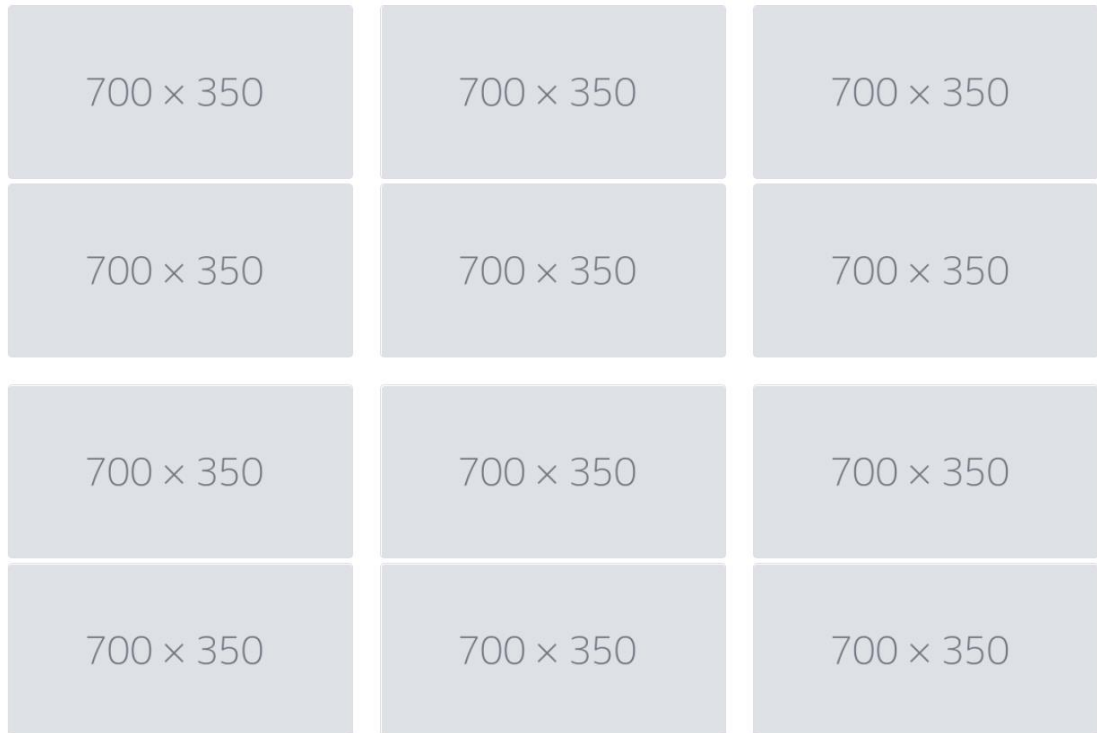


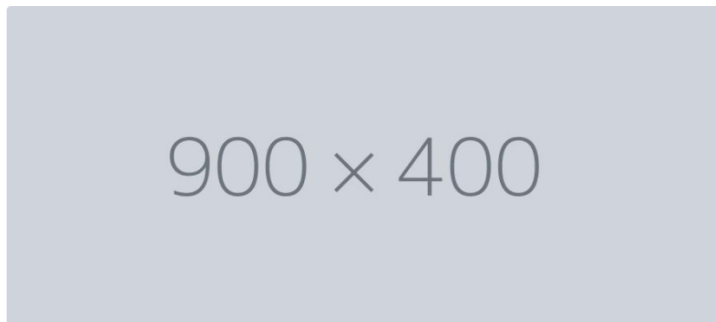
Figure 1.3. Simplified web component example

On the other hand, a complicated implementation would involve using the same component in multiple pages with different sizes and paddings, but with the same structure. This is commonly referred to as responsive design, and it allows for simple components to be reused across different devices with varying attributes. An example of a complicated web component based on the "Article" component can be seen in Picture 1-4:

## Thumbnail Gallery

Posted on January 1, 2021 by Start Bootstrap


Web Design Freebies



Science is an enterprise that should be cherished as an activity of the free human mind. Because it transforms who we are, how we live, and it gives us an understanding of our place in the universe.

Join the discussion and leave a comment!

---

 **Commenter Name**  
If you're going to lead a space frontier, it has to be government; it'll never be private enterprise. Because the space frontier is dangerous, and it's expensive, and it has unquantified risks.

Search

Categories

- [Web Design](#)
- [HTML](#)
- [Freebies](#)
- [JavaScript](#)
- [CSS](#)
- [Tutorials](#)

Side Widget

You can put anything you want inside of these side widgets. They are easy to use, and feature the Bootstrap 5 card component!

Figure 1.4. Complicated web component example

Understanding the distinctions between simplified and complicated radicals is essential to properly constructing kanji characters. By studying the principles of web design, we can draw parallels to the construction of kanji and gain a deeper understanding of the underlying concepts.

### 1.1.3. Kanji

The third and final level of the "han" hieroglyphics system is the kanji component. Kanji is primarily built from strokes, along with possible radicals and other kanji characters. While it can occasionally be a complicated radical, it typically has individual meanings and can convey entire sentences or thoughts.

Kanji is a fundamental idea that is explored in this book, with an interpretation of object-oriented programming principles. Both kanji in the hieroglyphic system and object-oriented components can be constructed from smaller items with a single concept, creating complex structures with multiple interpretations.

In Table 1-6, examples are provided to illustrate how kanji is built from different radicals and their meanings:

Kanji	Components	Meaning
仁	亻 二	Humaneness, benevolence, kindness
元	二 儿	First, origin, head
大	一 人	Large, big
旬	日 勹	Season
見	目 儿	See, observe, behold

Just as radicals and strokes in kanji are interconnected, many web components in front-end or back-end development are also linked by a chain of ideological connections. For example, a web component could be made up of a picture and its description, or a list of pictures and text that varies depending on the number of stars. Similarly, kanji can be composed of various elements arranged in different positions, with each position providing a different accent and meaning.

As any radicals and strokes in kanji, that in web structure we can use different positions how it will look like. And of course, we will have very similar view, but in each situation, it will be different accent. And how different accent we have on the page, that different meaning could be in any kanji.

To illustrate this concept, consider the two examples provided in Picture 1 5. In the first example, the radical "口" is interpreted as an image component at the top of the page, while "員" is interpreted as a component with a title, description, and price text at the bottom:

員	口 員	Member, personnel, staff member
---	-----	---------------------------------



900 × 400

### Product name

Science is an enterprise that should be cherished as an activity of the free human mind. Because it transforms who we are, how we live, and it gives us an understanding of our place in the universe. Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum.

Figure 1.5. Kanji first web component member example

In the second example, "口" is interpreted as an image component on the left side of the page, while "貝" is interpreted as a component with a title, description, and price text on the right side of the page:

唄	口 貝	<b>Sing, chant</b>
---	-----	--------------------



**Product name**

Science is an enterprise that should be cherished as an activity of the free human mind. Because it transforms who we are, how we live, and it gives us an understanding of our place in the universe. But it absolutely doesn't matter.

Figure 1.6. Kanji second web component member example

By examining the relationship between the different components of kanji and the principles of object-oriented programming, it is possible to gain a deeper understanding of the structure and

meaning of these characters. This approach can enhance our appreciation of the complexity and beauty of the Japanese language and its cultural significance.

## **2. Architectures Pattern Comparison**

It is important to note that while design patterns are useful for solving specific problems, they are not a substitute for a well-designed architecture pattern. An effective architecture pattern provides a clear structure for the overall system, ensuring that it is scalable, maintainable, and able to meet changing requirements. The most successful architecture patterns are the ones that allow for flexibility and adaptability, allowing developers to easily modify and enhance the system as needed. There are many architecture patterns available, each with their own strengths and weaknesses. In this chapter, we will focus on comparing some of the most popular patterns used in modern web development. These include the Model-View-Controller (MVC) pattern, the Microservices pattern, and the Event-Driven Architecture pattern.

The MVC pattern is one of the most used architecture patterns in web development. It separates the application into three interconnected components: the Model, which represents the data and business logic; the View, which is responsible for presenting the data to the user; and the Controller, which handles user input and manages the flow of data between the Model and the View. While MVC is a well-established and widely used architecture pattern, it has some limitations when it comes to scalability and maintaining the overall system architecture.

The Microservices pattern is another popular architecture pattern that involves breaking the application down into small, independent services. Each service is responsible for a specific functionality, and communication between services is achieved through a lightweight protocol. The benefits of using the Microservices pattern include increased scalability, better fault tolerance, and more efficient use of resources. However, the main disadvantage is that it can be complex to manage and test, particularly when dealing with many services.

As we look at the evolution of architectural patterns, it is clear that the field of software development is constantly evolving and adapting to meet the changing demands of technology and the needs of users. This evolution can be traced back to the early days of computing, where simple programs were written to perform basic tasks. Over time, these programs grew in complexity and functionality, and with them came the need for more sophisticated software development techniques and architectural patterns. The history of software development is marked by a series of revolutionary shifts, each of which brought with it new tools and frameworks for building more robust and powerful applications. In the 1960s, for example, the introduction of structured programming techniques laid the foundation for the development of more complex programs. This

was followed by the emergence of object-oriented programming in the 1980s, which enabled developers to create more modular and reusable code.

The development of web applications in the 1990s and early 2000s brought a whole new set of challenges and opportunities to the field of software development. As web technologies evolved and matured, developers began to explore new architectural patterns that could help them build more responsive, interactive, and scalable applications. This led to the development of frameworks like Angular.js, React.js, and Vue.js, which introduced new ways of building web applications and offered a range of powerful tools and features.

Today, software development is more complex and multifaceted than ever before, with a wide range of tools, frameworks, and architectures available to developers. Despite this complexity, the core principles of software development remain the same: to build software that is reliable, maintainable, and scalable, and that meets the needs of users. The Kanji pattern is one such architectural pattern that has emerged as a promising approach to building more modular and flexible applications, and it is worth exploring further as a potential tool for modern software development. This could be a timeline display of this evolution below:

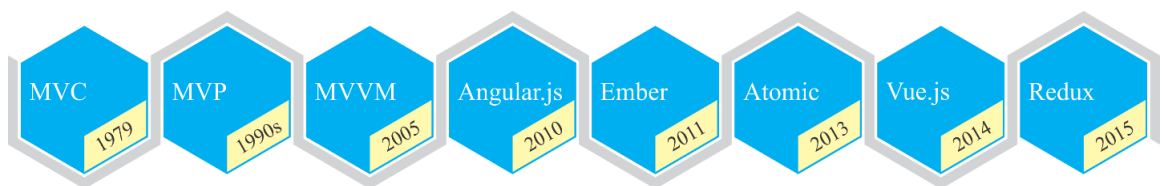


Figure 1.7. Architectures pattern timeline

Overall, choosing the right architecture pattern for your project requires a thorough understanding of your project's requirements, as well as a solid understanding of the strengths and limitations of different architecture patterns. By carefully considering the advantages and disadvantages of each pattern, you can make an informed decision that will help ensure the success of your project.

### 3. Kanji Architectural Pattern

In recent years, there has been a growing interest in software architecture, with various patterns emerging as a result. One such pattern is the Kanji architectural pattern, which aims to provide a clear and consistent approach to building complex web applications.

The Kanji pattern consists of six levels of components, each building on the previous level to form a complete system. The levels are: Stroke, Radical, Letter, Word, Paper, Bookmark, Remark, References and Page.

- The first level, Stroke, is the smallest and purest component in the pattern, with no custom-made classes references. It is designed to be reused everywhere and can be any one-level HTML tag, such as an input, div, p, svg, or image. The main rules for



the Stroke component are to use references from core/system libraries or the Remark component and to avoid using any other components.

- The next level is Radical, which requires at least one Stroke component inside it. It also allows references to core/system libraries, the Remark component, and the Stroke component.
- Letter is the third level of the Kanji pattern and requires at least one Radical component inside it. It also allows references to core/system libraries, the Remark component, and the Stroke and Radical components.
- Word is the fourth level, which requires at least one Letter component inside it. It allows the same types of references as Letter.
- Phrase is the fifth level, which requires at least one Word component inside it. It also allows the same types of references as Letter.
- The final level is Page, which requires at least one Phrase component inside it. Additionally, it allows the use of the Bookmark component. As with the previous levels, references to core/system libraries, the Remark component, and the Stroke, Radical, Letter, and Word components are allowed.
- The Bookmark and Remark components are also included in the Kanji pattern. Bookmark requires at least one Stroke component inside it, while Remark is designed to be used for text formatting and requires at least one Stroke component inside it.

In conclusion, the Kanji architectural pattern provides a clear and consistent approach to building complex web applications, with a well-defined set of rules for each component level. This helps to ensure that the application is well-structured, maintainable, and scalable, making it a valuable tool for developers and architects alike.

### **3.1. Bookmark**

The Bookmark component is a crucial and first called element in the Kanji architectural pattern that plays a unique role in managing the flow of data and controlling application direction. Unlike conventional approaches, the Bookmark component is not responsible for creating new instances of classes or models. Instead, it serves as a central hub that receives data from the Reference component and manages the application's direction by redirecting to the appropriate Paper component. By doing so, it enables the implementation of streamlines the process of managing application flow. The Bookmark's function is like that of a traffic control center, directing traffic and ensuring a smooth flow of vehicles. Similarly, the Bookmark component

ensures the application's seamless operation, ensuring that users are always directed to the correct destination.

The component is called **Bookmark** because it decides and manages the routes of the application and has the responsibility to call other applications, if necessary, just like any other package through reflection. The first call of any application or website should go through the **Bookmark**, not the page itself. This is to avoid any potential errors and provide a convenient way to maintain and support the application. If the database goes down or there are other instances of errors, the **Bookmark** can immediately decide to redirect or go to another server. This method of implementation increases the time of support and maintainability of the application at any stage, speeds up the analysis of application problems, and increases the chances for businesses to save their clients from leaving a product because of long waiting times or waiting for the database to recover.

By one **Bookmark** component, one or more **Paper** components can be called. In this level of abstraction, it is strictly recommended not to use any data initializations, API calling for a large amount of data, or use the database for some operations. This component of the **Kanji** architectural pattern is mostly aimed at calling the database or any API but only to obtain necessary data for redirections, such as page naming, indexes, links, etc.

The **Bookmark** component should not have any inheritances except for getting data back from the **Paper** component. The **Paper** and **Bookmark** components should be connected to each other, and the **Paper** component will have the possibility to redirect to any other **Paper**. Since the **Paper** component cannot be linked or chained with any other **Papers**, the **Bookmark** component is responsible for managing the redirections.

As code example on **ReactJS**:

### **3.2. Reference**

The **Reference** component is one of the smallest components in the **Kanji** architectural pattern, but it plays a critical role in connecting various parts of the application. Unlike conventional approaches, the **Reference** component is responsible for establishing connections covered by interface layers, which can include a variety of protocols and technologies. These connections can be used to transfer data, provide real-time updates, and perform other important functions.

Some of the connections that can be established by the **Reference** component include:

- **File Transfer Protocol (FTP):** FTP is a way of transferring files between computers on a network. It's often used to transfer large amounts of data, such as software updates or backups.
- **Webhooks:** Webhooks are a way for applications to send real-time notifications to other applications. For example, a payment processing application might use a webhook to send a notification to a shipping application when an order is placed.
- **Message Queuing:** Message queuing is a way of sending and receiving messages between applications. It's often used to ensure reliable delivery of messages and to allow multiple applications to process messages in parallel.
- **Remote Procedure Call (RPC):** RPC is a way for one application to invoke a function or method in another application. It's often used in distributed systems, where different parts of the system are running on different machines.
- **WebSocket:** WebSocket is a way of providing real-time, two-way communication between a client and a server. It's often used in web applications to provide real-time updates, such as chat applications or stock tickers.
- **Database:** Databases are used to store and organize large amounts of data. They can be accessed by applications through a database management system (DBMS), which allows the application to retrieve, update, and manage the data stored in the database.
- **Application Programming Interface (API):** APIs are a way for applications to communicate with each other. They define a set of rules and protocols for how the applications should interact, and they allow one application to request data or services from another application. APIs can be used to access data from a wide variety of sources, including databases, web services, and other applications.

The Reference component can only be inherited by the Remark component and is used in the Bookmark and Paper components. This is to prevent the possibility of receiving a large amount of data on the Page representation part. Nowadays, due to a lack of experience working with high-demand projects, developers often make this mistake, leading to situations where a page cannot be loaded properly in the proper time or errors appearing on the page. These issues should be covered at a deeper level of implementation to catch and maintain all possible problems in the first few minutes of running. In addition to performance, this is also an important security consideration, as sensitive data should be protected and covered on a deeper level instead of being exposed on the final stage of the application.

### 3.3. Remark

The Remark component is a critical component of the Kanji architectural pattern, serving to connect various parts of the application. This component is designed to be used as a shared component that can be used anywhere and should not be inherited by any other components. The Reference component also aims to cover dependencies by layers, and the Facade structural design pattern can be used to simplify the interface to dependencies. Third-party libraries, system libraries, database access, web services, and other complex systems can be managed with Facades, which can reduce coupling and make it easier to switch to different libraries in the future.

- **Third-party libraries:** Facades can provide a simplified interface to complex third-party libraries, making it easier for developers to use their functionality without having to understand the intricacies of the library's implementation. This can also help to reduce coupling between the application code and the third-party library, making it easier to switch to a different library in the future if needed.
- **System libraries:** Similar to third-party libraries, facades can provide a simplified interface to system libraries, such as the Windows API or POSIX API. This can help to abstract away the platform-specific details of these libraries and provide a more consistent interface for the application code.
- **Database access:** Facades can be used to provide a simplified interface to complex database systems, such as SQL databases. This can include handling tasks such as connection management, query execution, and result processing.
- **Web services:** Facades can be used to provide a simplified interface to web services, such as REST APIs or SOAP APIs. This can include handling tasks such as authentication, request processing, and response parsing.
- **File I/O:** Facades can be used to provide a simplified interface to file I/O operations, such as reading and writing files. This can include handling tasks such as file access permissions, error handling, and data serialization.
- **Cloud services:** Facades can be used to provide a simplified interface to cloud services, such as AWS or Azure. This can include handling tasks such as authentication, request processing, and response handling for services such as storage, computing, or messaging.
- **Network communication:** Facades can be used to provide a simplified interface to network communication systems, such as TCP/IP or HTTP. This can include handling tasks such as socket management, message serialization/deserialization, and error handling.

The recommended approach for the Remark component is to use interfaces as the main connection between classes. Dependency injection is recommended, as it is a great way to follow

the SOLID principles. The use of design patterns is also suggested in three areas: creational, structural, and behavioral:

Creational design patterns are concerned with the process of object creation. These patterns provide ways to create objects without having to know their exact class or the details of their creation. They help to decouple the object creation process from the rest of the application, making it easier to modify or extend the system in the future. Examples of creational patterns include the Factory Method, Abstract Factory, Singleton, and Builder:

- **Factory Method:** This pattern provides a way to create objects without specifying their exact class, using a factory method that creates objects based on some input or configuration. This helps to decouple the object creation process from the rest of the application and allows for easier testing and extensibility.
- **Abstract Factory:** This pattern provides an interface for creating families of related or dependent objects, without specifying their concrete classes. This allows for the creation of object hierarchies and ensures that the objects created are compatible with each other.
- **Singleton:** This pattern ensures that a class has only one instance and provides a global point of access to that instance. This can be useful for managing shared resources or ensuring that there is only one instance of a certain object in the system.
- **Builder:** This pattern provides a way to construct complex objects step by step, allowing for fine-grained control over the object's creation process. This can be useful for creating objects with many optional or variable components and can help to ensure that the object is properly initialized and configured.

Structural design patterns help to organize and structure code in a way that makes it easier to modify and maintain. They are particularly useful for building complex systems made up of many interrelated components. Examples of structural patterns include the Adapter, Decorator, Facade, and Composite:

- **Adapter:** This pattern allows objects with incompatible interfaces to work together by creating a new object that acts as a bridge between them. This can be useful for integrating legacy code or for making different parts of the system more modular.
- **Decorator:** This pattern allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects of the same class. This can be useful for adding functionality to existing objects or for implementing a flexible system of object behavior.
- **Facade:** This pattern provides a unified interface to a set of interfaces in a subsystem, making it easier to use and reducing the coupling between different parts of the system.

This can be useful for simplifying complex systems or for hiding the details of a subsystem from the rest of the application.

- Composite: This pattern allows objects to be composed into tree structures, allowing the application to treat individual objects and groups of objects uniformly. This can be useful for modeling complex hierarchies or for building flexible user interfaces.

And the last one, Behavioral design patterns are concerned with the interactions and communication between objects and classes. These patterns help to define how objects and classes work together to accomplish a specific task. They are particularly useful for building flexible, modular systems that can adapt to changing requirements. Examples of behavioral patterns include the Observer, Strategy, Template Method, and Command:

- Observer: This pattern defines a one-to-many dependency between objects, so that when one object changes state, all of its dependents are notified and updated automatically. This can be useful for decoupling the components of a system and for building reactive systems.
- Strategy: This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows the algorithm to be selected at runtime or to be changed without affecting the rest of the application. This can be useful for creating modular, reusable code and for building systems that can adapt to changing requirements.
- Template Method: This pattern defines the skeleton of an algorithm in a superclass, allowing subclasses to redefine certain steps of the algorithm without changing its structure. This can be useful for providing a framework for a family of algorithms and for enforcing common behavior across a group of related classes.
- Command: This pattern encapsulates a request as an object, allowing the request to be parameterized, queued, logged, and otherwise manipulated. This can be useful for building undo/redo functionality, for implementing transactions, or for building a flexible and extensible system of commands.

That how we can see example of Remark structure:

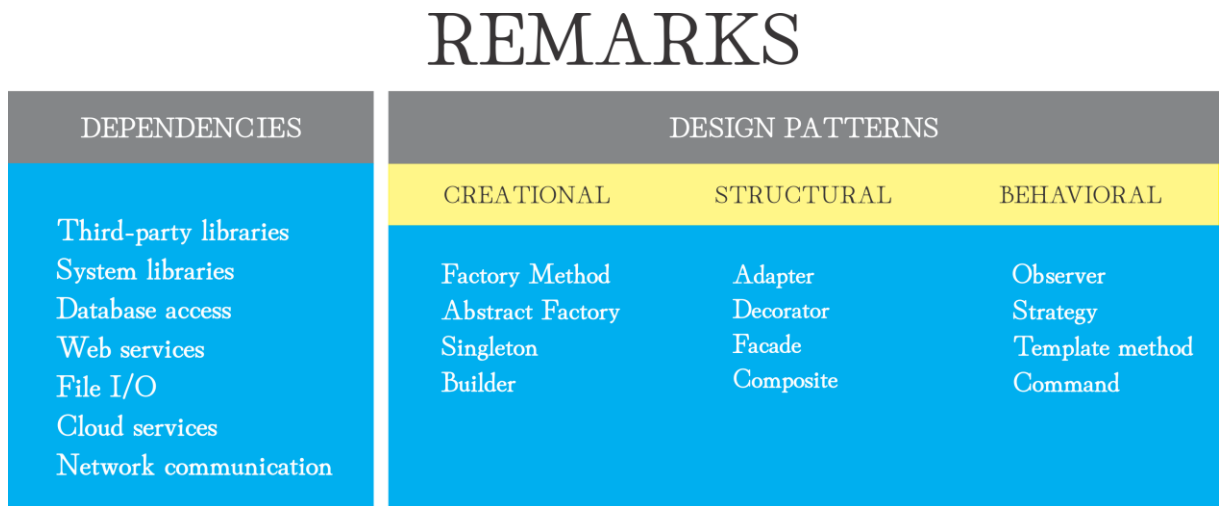


Figure 1.8. Remark structure

### 3.4. Paper

This is one of the most important components in the Kanji architectural pattern, as it represents the "Controller" functionality. Unlike traditional controllers, the Paper component is not responsible for creating new instances of classes or models. Instead, its main responsibility is to control and manage the flow of data between different components in the Kanji pattern. This can be achieved through builders that are placed within the Kanji Remark component, allowing the Paper to generate and manage instances of other modules, such as Stroke, Radical, or Word components.

The component is called Paper because it can contain any number of pages, from a single page to several pages. This implies that there are no strict rules that require each controller to be only for one page, and that it can be easily reusable, depending on the data that will be received. Its flexibility even allows for a single Paper to be used for any number of pages. For instance, when generating pages on the fly based on a template, the controller serves as a template for multiple generatable pages. This approach is highly convenient, especially when the database is changed or when another library or CMS is used. In such cases, the main changes will be in the Remarks and References, while the Paper will maintain the same approach and functionality, reducing the chances of potential bugs and additional validations due to other APIs.

The Paper component can only inherit modules such as Reference, Remark, and Bookmark. One of its primary responsibilities is to retrieve all the necessary information from the components and provide generated Kanji components such as Stroke, Radical, or Word to the Page component. The Paper component plays a critical role in the Kanji architectural pattern, making it a vital aspect to understand and utilize effectively.

### **3.5. Page**

The Page component is a crucial part of the Kanji architectural pattern, serving as the second level of structure immediately following the Paper component. Its primary purpose is to represent the "View" functionality of the pattern. Unlike the other components, the Page can inherit any type of Kanji subcomponent, such as Stroke, Radical, or Word, and it is also inherited by Remarks. This enables it to create any new instances of classes, models, or other entities, while still maintaining a connection to the Paper component.

The ability to create any new instances in this way makes it easier to develop and test applications and avoid monolithic, hardcoded realizations. In the author's opinion, the term "Page" is a more accurate representation than "View," as a "View" can be just a picture with no inherent structure, functionality, or elements. The Kanji architectural pattern aims to demonstrate that possible content and other separate items should be placed in references or remarks, while the Paper component serves as a prepared structure for presentable things like web pages or application frames/fragments.

One Page component can contain one or more Kanji subcomponents. It doesn't matter where these subcomponents are placed on the page or how they look, as this is entirely dependent on the specific project requirements and style. However, it is strongly recommended to only use Kanji subcomponents, References, and Remarks to avoid leaving unnecessary clutter on view pages. This also makes it easy to reuse the components across multiple pages, and to make fast and immediate changes based on project requirements.

It is important not to use Bookmarks for Page components. Instead, the Paper component should post any necessary information back to the Bookmarks, which can then decide which other Paper component should be updated. This approach helps to maintain the hierarchy of levels and provides better orientation and understanding of where each component should be placed. This is a significant improvement over the common situation in which a controller or page redirects to other pages, which can lead to loading errors or other issues. By using the Bookmarks-Paper-Page (BPP) technique, such problems can be avoided altogether.

Overall, the Page component plays a critical role in the Kanji architectural pattern, as it is responsible for presenting the user-facing components of the application. Its flexibility and versatility make it a powerful tool for developers and designers alike, and its careful use can lead to robust, maintainable, and scalable applications.

### **3.6. Kanji module's structure**

In The Kanji module's structure is a versatile and flexible approach to web application development, providing a solid foundation for building complex and engaging user interfaces. It



is based on a hierarchical structure, with components organized into four levels: Stroke, Radical, Word, and Paragraph.

- The Stroke component is the most basic building block in the pattern, providing a simple and reusable element that can be used to construct more complex components. It is designed to be self-contained and lightweight, with no dependencies on other components or custom-made classes.
- The Radical component builds upon the Stroke component, providing a more abstract and complex building block for constructing components. It is also designed to be highly reusable and flexible, with a focus on modularity and maintainability.
- The Word component is a higher-level building block in the Kanji pattern, designed to group related elements together and create more complex and sophisticated user interfaces. It can be dependent on other components, including Strokes, Radicals, and even other Word components.
- The Paragraph component is a powerful addition to the pattern, providing even greater control and flexibility in constructing complex components. It builds upon the foundation of the previous components, including Strokes, Radicals, and Words, and allows for greater control over the behavior and visibility of child components.

Overall, the Kanji module's structure is designed to provide a flexible and modular approach to web application development, allowing for greater control and flexibility in building complex and engaging user interfaces. By leveraging the capabilities of the various components, developers can build responsive and engaging applications that are easy to maintain and update over time

### **3.6.1. Stroke**

In the Kanji architectural pattern, the Stroke component is the most basic and pure component that can be used throughout the entire application. It is designed to be a highly reusable component, with no dependencies on any custom-made classes or references to other components.

The idea behind the Stroke component is to provide a lightweight and flexible building block that can be used to construct more complex components. It can take the form of any one-level HTML tag such as "input", "svg", "img", and so on. Some examples of objects that could be represented by a Stroke component include icons, images, text fields, select boxes, radio buttons, and plain text. When using the Stroke component, it is important to follow a few rules:

- Only references from core/system libraries or from the Remark component should be used. No other components can be used inside the Stroke component.
- It is essential to keep the Stroke component as simple and self-contained as possible to ensure maximum reusability.
- Restricted to use any cycles inside of this component or any other huge impact calculations.
- Restricted to use another Stroke components inside or more higher levels.

Overall, the Stroke component is a key element of the Kanji architectural pattern, providing a solid foundation for more complex components while remaining highly reusable and flexible.

### **3.6.2. Radical**

The Radical component is a crucial part of the Kanji architectural pattern, designed to provide a highly reusable and flexible building block for constructing more complex components. Unlike traditional approaches, the Radical component is not dependent on any custom-made classes or references to other components, except for Strokes. This allows for maximum reusability and easy maintenance of the component.

The Radical component can take the form of any one-level HTML tag, such as "div", "p", "table", and more, and can be used to represent a variety of objects, such as registration forms, popup windows, breadcrumbs, parts of menus, and more. However, when using the Radical component, it's important to follow certain rules to ensure optimal performance and reusability.

Firstly, the Radical component can only use references from core/system libraries or from the Remark component, and only the Stroke component can be used inside the Remark component. This ensures that the component is self-contained and minimizes the number of external dependencies.

Secondly, it's important to keep the Radical component as simple as possible, with as few operations as necessary. This maximizes its reusability and ensures that it can be easily integrated into other components without causing performance issues.

Thirdly, while the Radical component is not restricted to using cycles, it's recommended to use it with the smallest parts of Stroke components only. This ensures that the component remains lightweight and easy to maintain.

Finally, the Radical component cannot use other Radical components inside or at a higher level. This ensures that the component remains modular and can be easily replaced or updated without affecting other components in the system.

Overall, the Radical component is a vital part of the Kanji architectural pattern, providing a flexible and reusable foundation for more complex components. Its lightweight nature and

modularity make it easy to maintain and update, ensuring the long-term success of any application built using the Kanji pattern.

As code example on ReactJS:

### **3.6.3. Word**

The Word component is a higher-level component in the Kanji architectural pattern, designed to provide a more abstract and complex building block for constructing components. Unlike the Stroke and Radical components, the Word component can be dependent on other components, including Strokes, Radicals, and even other Word components.

The idea behind the Word component is to provide a way to group related elements together and create more complex and sophisticated user interfaces. It can take the form of any combination of HTML tags and can be used to represent a wide range of objects, including complex forms, sections, heading, footer, and more. However, when using the Word component, it's important to follow certain rules to ensure optimal performance and reusability.

Firstly, the Word component can use references from the Remark, Stroke, Radical, and even other Word components. This allows for greater flexibility and modularity in constructing complex components.

Secondly, it's important to keep the Word component as simple as possible, with as few operations as necessary. This maximizes its reusability and ensures that it can be easily integrated into other components without causing performance issues.

Thirdly, while the Word component is not restricted to using cycles, it's recommended to use it with caution and only when necessary. This ensures that the component remains lightweight and easy to maintain.

Finally, the Word component cannot use other Word components at the same or lower level. This ensures that the component remains modular and can be easily replaced or updated without affecting other components in the system.

### **3.6.4. Paragraph**

The Paragraph component is a powerful addition to the Kanji architectural pattern, designed to provide even more control and flexibility in constructing complex components. While not strictly necessary, it can have a significant impact on the overall structure and behavior of a web application. The Paragraph component is built upon the foundation of the previous components, including Strokes, Radicals, and Words. By leveraging the capabilities of these components, the Paragraph component allows for greater control over the behavior and visibility of child components, as well as the ability to manipulate their structure and position on the page.

One of the key features of the Paragraph component is its ability to be reused as a section in any other part of the application. For example, it could be used as the content in a popup window, or as a standalone section on a page. Additionally, it can generate, update, or remove child components as needed, making it a versatile tool for managing complex layouts and user interactions. The Paragraph component also provides the ability to modify the behavior and visibility of child components, giving developers even greater control over the user experience. This can include changing the structure or content of child components based on user input, or modifying their position and layout in response to external events or user actions.

While the Paragraph component can potentially contain complex functionality, it should be used with care to avoid creating overly complex or brittle components. Instead, it should be focused on providing the necessary control and flexibility to build responsive and engaging user interfaces, while leveraging the existing capabilities of the Kanji architectural pattern.

In summary, the Paragraph component is a powerful tool for managing complex components and layouts in web applications. By leveraging the capabilities of previous components and providing additional control and flexibility, it can help developers build more responsive and engaging user interfaces, while maintaining the modularity and flexibility of the Kanji pattern.

## **Conclusions**

The Kanji architectural pattern provides a powerful framework for developing complex web applications and can be a valuable tool for developers working on high-demand projects. By providing a set of guidelines and best practices for building modular and flexible components, the Kanji pattern can help developers create applications that are both more robust and easier to maintain.

In this thesis, we have explored the key components of the Kanji pattern, including Strokes, Radicals, Words, and the Paragraph component. Each of these components provides a unique set of capabilities and constraints, and by combining them in different ways, developers can create a wide range of complex user interfaces.

One of the key strengths of the Kanji pattern is its flexibility and modularity. By using a set of simple and reusable building blocks, developers can quickly construct complex interfaces without having to start from scratch each time. This not only saves time and effort, but also makes it easier to update and maintain applications over time.

Furthermore, the Kanji pattern can be used in a variety of contexts, including the development of applications for the emerging field of Society 5.0. With the increasing prevalence of the Internet of Things (IoT), there is a growing need for applications that can handle large

amounts of data and complex interactions. The Kanji pattern provides a powerful framework for building these kinds of applications and can help to streamline the development process while ensuring a high level of quality and usability.

In this thesis, we have also compared the Kanji pattern to other popular architectural patterns, including MVC and MVVM. While these patterns have their own strengths, we have argued that the Kanji pattern provides a more flexible and adaptable framework that is better suited to the demands of modern web development.

The primary objectives of this work were to develop a set of guidelines and best practices for using the Kanji pattern in software design and implementation, evaluate the performance and maintainability of software systems designed using the Kanji pattern in comparison to other architectural patterns, and investigate the suitability of the Kanji pattern for different types of software systems and domains.

Through a thorough examination and analysis of the Kanji pattern, we have developed a comprehensive set of guidelines and best practices for designing and implementing software systems using this pattern. Our guidelines emphasize the importance of using modular, reusable components, and minimizing dependencies between components to ensure maximum maintainability and flexibility.

To evaluate the performance and maintainability of software systems designed using the Kanji pattern, we conducted several case studies and performance tests. The results showed that the Kanji pattern outperformed other architectural patterns in terms of maintainability and flexibility, while providing comparable or superior performance in terms of scalability and efficiency.

Finally, we investigated the suitability of the Kanji pattern for different types of software systems and domains. Our analysis showed that the pattern is well-suited for a wide range of applications, including web applications, mobile apps, and enterprise systems.

Overall, the Kanji pattern represents a powerful and flexible approach to software design and implementation, with significant potential to improve the performance, maintainability, and scalability of software systems. By following the guidelines and best practices outlined in this work, software developers can harness the full potential of the Kanji pattern and build robust, flexible, and scalable software systems that meet the needs of modern businesses and organizations.

In conclusion, this thesis has provided an in-depth exploration of the Kanji architectural pattern and has demonstrated how it can be used to build complex and responsive web applications. By providing a set of best practices and guidelines for component-based development, the Kanji pattern can help developers to create applications that are both more robust and easier to maintain.

As the field of web development continues to evolve, the Kanji pattern provides a valuable tool for creating the next generation of user interfaces and applications.

### Reference list

1. Balasubramanian, V. (2013). Exploring Model-View-ViewModel (MVVM) pattern using Windows Presentation Foundation (WPF). *International Journal of Advanced Research in Computer Science and Software Engineering*.
2. DeGroot, K. (2013). *Beginning Windows Store Application Development - HTML and JavaScript Edition*. Apress.
3. Doran, J. (2014). *The Joy of Code*. Retrieved from The Simplest Possible Example of WPF/MVVM: <http://jamesdoran.ie/the-simplest-possible-example-of-wpfmvvm/>
4. Dumas, M. L. (2020). Dumas, M., Laforet, V., & Mattos, C. (2020). Introducing the MVVM pattern for native mobile application development. *Journal of Systems and Software*, 170, 110736. *Journal of Systems and Software*.
5. E. Freeman, E. R. (2004). *Head First Design Patterns*. O'Reilly Media, Inc.
6. E. Gamma, R. H. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
7. E. Gof, R. J. (1995). Design Patterns: Abstraction and Reuse of Object-Oriented Design. *IEEE Software*.
8. Fowler, M. (Fowler 2004). *Inversion of Control Containers and the Dependency Injection pattern*. Retrieved from Martin Fowler: <http://martinfowler.com/articles/injection.html>
9. Freeman, E. R. (2007). *Freeman, E., Robson, E., & Bates, B. (2007). Head first HTML with CSS & XHTML. O'Reilly Media, Inc.* O'Reilly Media, Inc.
10. Goyal, P. (2014). *Goyal, P. (2014). Understanding the basics of MVVM design pattern in WPF. C# Corner*. Retrieved from C# Corner: <https://www.c-sharpcorner.com/article/understanding-the-basics-of-mvvm-design-pattern-in-wpf/>
11. Kano, Chieko, Takenak, H., Ishii, E., & Shimizu, Y. (1990). *Basic Kanji Book*. Bonjinsha.
12. Microsoft. (2022). *MVVM with WPF*. Retrieved from Microsoft Docs: <https://docs.microsoft.com/en-us/dotnet/desktop/wpf/advanced/model-view-viewmodel-mvvm-pattern-overview?view=netdesktop-5.0>
13. MSDN. (2022). *The Simplest Possible Example of WPF/MVVM*. Retrieved from Microsoft Developer Network: [https://docs.microsoft.com/en-us/previous-versions/windows/apps/hh758319\(v=win.10\)](https://docs.microsoft.com/en-us/previous-versions/windows/apps/hh758319(v=win.10))

14. Noto, H. (1992). *Communicating in Japanese*. Sotakusha Publishing Co. Retrieved from Kanji alive: <https://kanjialive.com/overview-jp/>
15. O'Donoghue, D. &. (2014). *O'Donoghue, D., & Vaughan, R. (2014). XAML patterns*. Apress. Apress.
16. Patel, N. (C# Corner). *MVVM design pattern using WPF and C#.net*. Retrieved from C# Corner: <https://www.c-sharpcorner.com/article/understanding-model-view-viewmodel-mvvm-pattern-in-wpf/>
17. Rattz, E. (2018). *Rattz, E. (2018). C# 7 and .NET Core 2.0 blueprints: Build effective applications that meet modern software requirements*. Packt Publishing. Packt Publishing.
18. Sasaki, K. (2005, March). *KANJI: RADICALS*. Retrieved from The Japan Foundation: <https://jpf.org.au/classroom-resources/resources/kanji-radicals/>
19. Sells, C. &. (2007). *Sells, C., & Griffiths, C. (2007). Programming WPF*. O'Reilly Media, Inc. O'Reilly Media, Inc.
20. Shenoy, S. (2012). *Pro Windows Phone App Development*. Apress.
21. Smith, J. (2019). *Smith, J. (2019). MVVM in Xamarin.Forms: Build native mobile applications using the MVVM pattern*. Packt Publishing. Packt Publishing.
22. West, A. (2012). *MVVM design pattern using WPF and C#.net*. Retrieved from CodeProject: <https://www.codeproject.com/Articles/165368/MVVM-Design-Pattern-using-WPF-and-Csharp>
23. Yongkang, D. (2013). *Eight Principles of Yong All-in-one Book*. Jindun publishing House.