Unstructured Programming Considered Harmful

Mats Danielson

Dept. of Computer and Systems Sciences
Royal Institute of Technology and Stockholm University
Electrum 230, SE-164 40 Kista, Sweden
mad@dsv.su.se

Abstract: Jackson Structured Programming (JSP) emerged in the mid-1970s as a refinement of structured programming, offering a data-driven design methodology uniquely tailored to the challenges of sequential input and output processing. While the broader structured programming movement succeeded in eliminating the chaos of unrestrained `GOTO`s, it offered little guidance on how to derive program structure from the specific data constraints of real-world tasks. JSP filled this gap by asserting that program control flow should be isomorphic to the hierarchical organisation of input and output data structures. Despite early traction and pedagogical success, JSP has largely faded from mainstream practice. This essay argues that this fading is unjustified: the problems JSP addressed have not disappeared, and the method remains as effective today as ever. The essay provides a historical account of JSP's emergence, explains its theoretical foundations and practical methodology, and highlights its enduring relevance. It connects JSP to finite state machines and parsing, demonstrating its alignment with automata-theoretic principles while also exploring program inversion for interactive applications. Drawing on illustrative examples and contemporary software development practices, the essay shows how neglecting JSP principles leads to convoluted, fragile, and expensive code, problems that persist in today's software landscape. Furthermore, it advocates for a modern resurrection of JSP ideas, not as a nostalgic revival, but as a necessary recalibration of software craftsmanship. In an era dominated by frameworks, libraries, and design patterns, this essay calls for a rebalancing toward clear, disciplined structuring of control flow guided by data. In doing so, it argues that JSP represents not a historical curiosity, but an underappreciated design philosophy with real, ongoing value in programming education and practice.

Keywords: Structured Programming, Program Design, Data-Driven Software Architecture, Finite State Machines, Software Maintainability, Control Flow Structuring

1. Introduction

In the late 1960s and early 1970s, amid a growing crisis of "spaghetti code" and unmaintainable software, the discipline of structured programming emerged as a remedy. Edsger W. Dijkstra's famous 1968 letter *Go To Statement Considered Harmful* summarised the call to eliminate arbitrary jumps and impose logical structure on programs. It did not, in fact, call for the elimination of `GOTO` use as has been subsequently claimed, and the title was not chosen by Dijkstra but by the editor of Communications of the ACM, Niklaus Wirth, to spice things up. By the early 1970s, academics and practitioners alike were advocating for programs to be composed of well-defined control constructs (sequence, selection, and iteration) instead of chaotic tangles of `GOTO` statements. That these three constructs were sufficient was proved by Böhm and Jacopini in 1966 in the most influential computer science article of all time. This *structured programming* movement, documented in works like *Structured Programming* (1972) by Dahl, Dijkstra, and Hoare, promised improved clarity, reliability, and ease of maintenance. It led to language features that encouraged structured flow and to methodologies for top-down program design. Within this post-`GOTO` era, Michael A. Jackson introduced a distinctive approach known as JSP. First described in his 1975 book *Principles of Program Design*, JSP built on the structured programming ethos but added its own powerful principle: that the structure of a program should be dictated by the structure of that program's input and output data. In other words, Jackson argued that a well-structured program is one whose control flow is isomorphic to the hierarchical organisation of the data it processes, an idea both remarkably intuitive and, as this essay will discuss, largely neglected in modern practice.

JSP is a concrete methodology aimed at improving the design of programs that read input files and produce output reports. While sharing the broader goals of structured programming (reducing complexity and improving maintainability), JSP pursued these goals through a unique data-structured approach to program design. In doing so, it addressed problems that the generic structured programming movement left open: how to choose an appropriate program structure for a given task, and how to ensure the program's logical structure aligns with the inherent structure of the data being handled. Jackson's answer was to analyse the input and output data streams and derive the program's architecture from them. As we will see, this approach yields programs that are often easier to understand and modify, because changes in data formats or requirements tend to map to localised changes in the corresponding program structure.

This essay examines JSP's position in depth, charting its rise, its core concepts, and its decline in usage. We will argue that JSP was an important (if now often forgotten) milestone in software engineering, one that introduced a fundamental design insight still relevant today. The discussion will cover how JSP aligns program control flow with data structures, how it relates to formal concepts like finite state machines, and what kinds of problems it was intended to solve. We will also consider how contemporary programming practice has drifted away from such methodical design approaches, often relying on intuition or ad hoc methods, and the consequences this has for software complexity and maintenance costs. Although JSP is not a panacea for all programming tasks (and Jackson himself delineated cases where it needs an extension or cannot directly apply), its core principles remain valuable. By revisiting JSP and reflecting on its enduring lessons, we can better understand the gap in today's prevalent practices and why reintegrating some of JSP's discipline with clear guidance on when and how to apply it could lead to better, more maintainable code. Before delving into JSP's specifics, we begin with the historical context in which it arose and the motivations behind its development.

2. Structured Programming

The 1960s saw software projects grow in size and complexity, often outpacing the techniques available to manage them. Early software was frequently written in assembly or unstructured high-level code, and it was common for programs to be judged by whether or not they ran, not by how well they were written. As systems grew, this lack of emphasis on code quality and structure led to what was pejoratively called "spaghetti code": programs with convoluted control flow that tangled jumps and logic in knots. Debugging and modifying such code proved nightmarishly difficult, contributing to the so-called software crisis of that era.

Structured programming emerged as a response to this crisis, promoting the idea that programs should be composed from a limited set of well-understood control structures (notably sequence, conditional selection, and loops with a single entry and exit) instead of arbitrary jumps. The structured programming thesis, crystallised by Dijkstra and others, held that any program logic could be achieved by nested structures of these constructs, making `GOTO` unnecessary. This discipline promised to make programs more comprehensible and proofs of correctness more feasible. By the early 1970s, the movement had gained mainstream traction: for instance, the language Pascal (designed by Niklaus Wirth) embodied structured programming principles by providing structured loops and conditionals while largely eschewing unrestrained jumps. However, structured programming in its extreme form mainly provided negative guidance (do not use `GOTO`) and a general methodological suggestion to build programs hierarchically. It said less about how to choose the right hierarchy for a given problem. Many developers still designed programs in an ad hoc manner, perhaps doing a rough top-down decomposition or simply following the flow of input and output operations as they imagined them. It was thought back then that professional programmers should first create a design using their own personal method and only afterward legitimise it by casting it into the shape of one of the approved methods. In other words, even with structured programming as the reigning paradigm, the actual process of designing program structure often remained informal and intuition-driven. The result could be code that technically avoided `GOTO`s yet still lacked a coherent guiding structure, especially in the realm of data processing where input/output formats and business rules added complexity.

It was in this climate that Michael Jackson developed Jackson Structured Programming. Jackson, a British software consultant working in the commercial data processing sector, recognised that simply removing `GOTO`s did not guarantee clarity or maintainability. A program with only nested loops and if-statements could still be poorly structured if those loops and conditionals were arranged arbitrarily. What was needed was a more concrete method to derive a logical structure for programs *based on the problem at hand*. In the domain of business data processing, Jackson observed, the problem at hand typically revolved around reading and writing data files with known formats (e.g. master files, transaction files, report layouts). The structures of these data files (the hierarchy of records, subrecords, and fields) were often complex, and a key source of difficulty was the mismatch between the data's structure and the program's structure. Traditional approaches in that era tended to default to very simple program structures. A common practice was to read an input file record by record in a single loop and handle special cases within that loop. Jackson asserted that this habitual one-big-loop program structure was in fact almost always wrong for non-trivial data. For example, consider a sorted input file where groups of consecutive records belong to the same category (such as transactions grouped by customer, or runs of identical entries that need to be counted). A conventional single-loop implementation would iterate through every record and use `if` statements to detect boundaries between groups and aggregate records within a group. Such a program works, but it tends to be littered with flags and special-case logic. For instance, code to handle the first record of a new group or to clean up after the last record of the file. These edge cases arise because the program's single-loop structure does not naturally fit the hierarchical grouping in the data. In Jackson's view, the program should instead have been structured to mirror the two-level nature of the input (groups and records), for example by using a nested loop: an outer loop per group and an inner loop for records within a group. By not doing so, the traditional program was fighting against the grain of its data. This led to contortions in the code (extra logic to manage state transitions that a two-loop structure would handle implicitly), thereby making the program more error-prone and harder to modify.

Jackson's critique fit into the broader culture of software engineering at the time: know the structure of your problem, and reflect it in the structure of your solution. His specific contribution was identifying the structure of input/output data as the backbone for program structure. This was a step beyond general top-down design; it was a targeted strategy for a class of problems. It was aligned with the structured programming movement in insisting on well-formed control structures, but it went further by providing a recipe for determining the shape of those structures for a given data-processing program. In the next section, we will explore the emergence of JSP in more detail, including how he formulated it and its relationship to other methods of the period.

3. The Emergence of JSP

JSP was developed in the early 1970s, drawing from practical experience in commercial software development. As stated above, Jackson introduced the method formally in 1975 via his book *Principles of Program Design*, which became the foundational text for JSP. JSP was conceived at a time when many programs would take in sequential files (often on magnetic tape or disk), process them, and produce sequential output files or printed reports, although many modern programs also read from e.g. a socket and produce output to e.g. a canvas. Jackson zeroed in on these kinds of programs, which had well-defined input and output data structures, and he sought to make them easier to modify and reason about. In a retrospective talk in 2001, Jackson explained that his aim was to improve the modifiability of programs by structuring them according to their data, a method that in principle could be used with any programming language that supports standard structured programming constructs.

When JSP was unveiled, it did not stand entirely alone; it had parallels to other contemporary methodologies. Notably, a similar approach was advocated by Jean-Dominique Warnier and later by Kenneth Orr. The Warnier/Orr method (originating in the 1970s as well) also emphasized designing programs based on the structure of data, especially output reports. The key difference was that Warnier/Orr tended to focus almost exclusively on output data structure, whereas JSP insisted on considering both input and output structures together. In effect, Jackson's method was a more comprehensive data-structured design: instead of structuring a program solely around the format of the output, it sought a program structure that could *simultaneously* reflect the organisation of all input files and output files that the program needed to reconcile. As we will discuss, this is a non-trivial extension that leads to the need for techniques to resolve mismatches between different data structures.

Initially, JSP gained traction, especially in the UK and Europe. It was taught in courses and used in industry for designing data processing programs. Several factors contributed to the spread, and later the fading, of JSP. On the positive side, JSP was appealing because it was teachable and relatively concrete. Unlike some abstract software design theories, JSP came with a step-by-step procedure and a notation (structure diagrams) that could be directly applied to real programming tasks. Many who learned it reported that for the first time, they understood how to design programs that they had been writing without really understanding them. This suggests that JSP filled a pedagogical gap: it gave programmers a tangible strategy for program design, not just coding. Additionally, tool support emerged to help implement JSP designs, which likely aided adoption in some corporate environments.

However, JSP also had its limits and timing challenges. By the mid-1980s, the software development world's attention was shifting. Interactive, event-driven systems were becoming more common (transaction processing systems, GUIs, real-time control, etc.), and the rise of object-oriented programming and design was on the horizon. Jackson responded to the need for handling interactive systems by developing Jackson System Development (JSD) in the early 1980s, which generalised JSP's ideas to concurrent systems of processes. JSD introduced the notion of modelling real-world entities as processes (which can be seen as a form of finite state machines) and used *program inversion* to derive runnable systems from those models (more on inversion later). While JSD extended the life of Jackson's methodologies into the realm of system design, the industry's mainstream gradually gravitated toward other approaches such as structured analysis and design methods from Yourdon, DeMarco, Gane & Sarson, and later widespread methods such as UML and object-oriented analysis. JSP, focused on program-level design for data processing, slowly fell out of mainstream use.

In retrospect, JSP's eclipse can be attributed partly to changing fashions in software engineering, and partly to the misconception that its principles were only relevant to old-fashioned programming. As this essay will argue, that is a misconception. The core ideas of JSP have much broader applicability. But historically, once structured programming had been accepted and subsumed into general knowledge, many educators and practitioners did not specifically teach JSP's data-driven structuring as a separate topic. It became a kind of lost art, remembered mainly by those who had been trained in it during its heyday. Nevertheless, JSP's influence can be discerned in certain quarters: for example, the idea of designing a program like a parser for its inputs is commonplace in compiler construction (where recursive descent parsing is essentially the same idea applied to grammars). Jackson himself acknowledged in 2001 that the JSP approach was essentially the relationship exploited in parsing by recursive descent and that some issues JSP encountered (like lookahead constraints) were well-known in formal language theory. In other words, JSP had unknowingly bridged to computer science concepts of grammars and automata, albeit developed independently from the formal academic work.

To summarise, JSP emerged as a pragmatic methodology in the 1970s, aiming to improve the design of structured programs by aligning them with data structures. It gained a following and contributed significantly to the toolbox of structured design methods, even as it remained most popular in certain regions and domains. Over time, its explicit practice dwindled, but its legacy endures indirectly. In the following sections, we delve into the substance of JSP: its core principle and method, how one applies it to design a program, how it deals with multiple inputs/outputs and the finite-state perspective, and why all of this matters for producing better programs.

4. Data-Structured Program Design

At the heart of JSP is one guiding principle: a program's structure should follow the structure of its data. More specifically, JSP prescribes that the control flow structure of a program (the sequence of operations, the loops, and the conditionals) should be derived from the hierarchical structure of the input and output data streams that the program processes. This principle may sound straightforward, but it was a departure from how programs were typically designed. In practice, it means that if the input data is logically organised into, say, files of records, and those records in turn have sub-structures, then the program should have correspondingly nested loops or subroutines to handle those files and sub-structures. Similarly, if the output has a certain grouped format, that grouping should be reflected in the program structure.

To illustrate this principle in action, consider a simple yet illuminating example that Jackson often used: an input file contains multiple customer groups, where each Customer Group consists of one Customer Record followed by some number of Order Records for that customer. Furthermore, suppose there are two kinds of orders: Simple Order and Urgent Order, distinguished by a flag in each order record. The task of the program is to read this file and perhaps produce some output per customer. What structure should the program have? According to JSP, the answer is clear: the program's structure should mirror the data structure. At a high level, the file consists of zero or more Customer Groups (since there could be many customers, one after the other). Each Customer Group in turn consists of one Customer Record followed by zero or more Orders, where each Order is either a Simple Order or an Urgent Order. We can express the input structure succinctly as a kind of formal pattern (in fact, a *regular expression*):

```
File = ( CustomerRecord, ( SimpleOrder | UrgentOrder )* )*
```

This notation says: a File is zero or more repetitions of a sequence that starts with a Customer-Record and is followed by zero or more of either a SimpleOrder or an UrgentOrder record. In a JSP diagram, this same structure would be drawn as a tree with nodes for File, Customer Group, Customer Record, Customer Group Body (the orders), etc., marking iterations with an asterisk and alternatives (Simple vs Urgent) with a circle. The critical point is that *the program should have exactly this structure as well*. In pseudocode form, the program would look like:

```
while (another customer group) {          // Process each Customer_Group
    process Customer_Record;
    while (another order in this group) { // Process each Order
        if (order is Simple) {
            process Simple_Order;
        } else {
            process Urgent_Order;
        }
    }
}
```

This nested loop and conditional structure correspond one-for-one with the data layout. The outer loop iterates over customer groups, the inner loop iterates over orders within a group, and the `if` distinguishes the two order types. Such a program would naturally handle the boundaries between groups (when one group's orders end, the outer loop cycles) and would require no special flags to detect when the last record of a group or file is reached (aside from the normal loop conditions). The first record and last record of each group are not special-cased in the code; they arise implicitly from entering or exiting the loops. This is exactly what Jackson meant when he said the program should embody the data structure: *the execution sequence of the program's parts mirrors the sequence of records and groups in the file.*

By structuring the program in this way, we achieve a form of clarity and alignment that has immediate benefits. If tomorrow the requirements change such that a new kind of Order record (say, a VIP Order) is introduced, we know where that change fits: it is another alternative under the Order-handling conditional. If instead, the input format changes such that orders are grouped differently, that would reflect in a different nesting, and again the program structure would be adjusted correspondingly. Jackson noted that in well-designed business programs, requirement changes are usually minor tweaks to the existing structures, and by having the program's structures match the input/output structures, small changes to the inputs and outputs should translate into small changes to the program. In contrast, if the program were not structured according to the data, for example if we tried to handle all customers and orders in one flat loop, then a change in data structure (like adding a new level or variant) could necessitate invasive changes scattered throughout the code.

The JSP philosophy thus yields programs that are data-driven in their architecture. This is a particular flavour of structured programming: one could think of it as *data-structured programming*. It goes beyond the generic advice of using loops and subroutines; it tells us which loops and subroutines to use and how to nest them. The method to arrive at this structure is straightforward in concept:

1. *Analyse the Input and Output Data Structures*: List out the hierarchical composition of each input file and each output report/file that the program deals with. These can be represented as structure diagrams or similar tree/grammar notations.
2. *Derive a Program Structure that Accommodates All Data Structures*: This is a creative and sometimes challenging step. The goal is to find a single program flow that can process all required inputs and outputs coherently. Often it starts from one dominant input's structure, then integrates others.
3. *Refine the Program Structure into Pseudocode and Code*: Once the structure (the skeleton of loops and conditionals) is decided, one can place the actual processing statements (read, write, compute, etc.) into the appropriate parts of that skeleton.

JSP provided notations to assist with these steps. The primary notation was the Jackson Structure Diagram, a tree-like diagram annotated with symbols for sequence, iteration, and selection. In the earlier example, the structure diagram of the file is a tree with root File, a child Customer Group (with an iteration symbol * indicating there are many), which in turn has two children: Customer Record (single) and Customer Group Body (the orders, marked with * as there are many orders). Under the Customer Group Body node are two children for the two order types, marked as alternatives (often with a small circle or some notation to indicate a choice). Jackson's structure diagrams essentially visualise the same information one might put in a BNF grammar or a regular expression for the data. It was a semi-formal way to capture the data and program structure on paper during design. One of JSP's notable contributions is that it treats the program structure and data structure as two manifestations of the same underlying tree. This duality is repeatedly emphasised in JSP literature. One can take a structure diagram and read it in two modes:

- *As a data description*: A File consists of a sequence of Customer Groups; each Customer Group consists of one Customer Record followed by a Customer Group Body; the Customer Group Body consists of a sequence of Orders; each Order is either Simple or Urgent….
- *As a program description*: The Program has a component that processes the File (perhaps implemented as a loop over customer groups). That contains a sub-component that processes a Customer Group (one iteration of the outer loop). Inside that, first execute logic to process a Customer Record, then execute a sub-component to process the Customer Group Body (which will be an inner loop over orders). Within that inner loop, for each order, choose either the Simple Order processing path or the Urgent Order processing path.

By ensuring these two descriptions are mirror images, JSP achieves what one might call *structural congruence* between data and program. The benefits are conceptual simplicity and traceability: one can trace each piece of input data to a corresponding part of the program. This dramatically reduces the mental effort to understand what the program does with the data since a reader can literally follow the structure.

It is worth noting that JSP was not the only methodology to argue for reflecting the problem structure in the solution. Top-down stepwise refinement (advocated by Wirth and others) also told programmers to break problems into subproblems hierarchically. The difference is that JSP gives a very concrete criterion for identifying the substructures: use the data streams as the guide. In data-centric business applications, this is a natural fit because the inputs and outputs often *define* the problem requirements. If one knows the format of, say, a bank's transactions file and the desired format of a summary report, that goes a long way toward understanding what the program must do. JSP leverages this by saying the format (structure) itself is the key to the program design.

In summary, the fundamental JSP principle of data-structured program design leads to programs that are essentially *homomorphic* to their inputs and outputs. This principle, while conceptually simple, was a major innovation in the context of the 1970s, and even today it stands as a reminder that the shape of our data should influence the shape of our code. In the next subsection, we will discuss the JSP design methodology. How one systematically goes from data structures to a program structure, and in doing so, we will encounter the concepts of multiple data streams, correspondences, and some of the complications that arise (such as structure clashes and lookahead issues). These issues will deepen our understanding of both the power and the limits of JSP.

4.1 The JSP Design Methodology in Practice

Designing a program using Jackson Structured Programming is a step-by-step process that ensures the final program structure corresponds to the data structures. Here we outline the typical procedure and elaborate on key aspects, such as handling multiple input/output streams and placing actual processing logic into the structure.

1. Identify and Describe Data Structures:

The first step in JSP is to identify all sequential data streams that the program will read or write. This might include one or more input files or data streams and one or more output files or reports. Each of these has an inherent structure, often defined by record types and their grouping. For example, an input file might contain records of types A, B, and C in some nested arrangement. An output report might have headings, detail lines, totals, etc. The JSP designer documents each data structure, usually as a structure diagram or equivalent textual schema. It is crucial at this stage to understand how the structures correspond. Often, the program's purpose is to transform input data into output data, so there will be logical relationships between parts of the input and parts of the output. For instance, an output summary line might correspond to a group of input transaction records.

2. Construct a Preliminary Program Structure

Using the data structure of one stream as a starting point (commonly the principal input file), the designer sketches a program structure that can traverse that data. This structure will be a skeleton of loops (iterations), sequences, and conditionals (for alternatives) reflecting that data's hierarchy. If there were only one data stream, the program structure could exactly match it. However, most interesting programs have to deal with multiple streams (e.g., reading two files and producing one output). Therefore, JSP requires merging multiple data structures into a single program structure. The rule is that the program structure should reflect all the stream structures, not just one. In practice, this means the program structure must be a sort of superimposition of the input and output structures. The designer looks for a way to interleave the processing of different streams in one coherent flow.

3. Check for Structure Clashes and Correspondence Problems

Once a candidate program structure is formed, JSP encourages the designer to check it against all data streams to ensure it indeed covers them properly. Sometimes, one finds that no single structured flow can cleanly accommodate all required inputs and outputs. Jackson identified common types of structure clashes that can occur. For instance, one file lists events chronologically and another needs them grouped by category (which might intermix chronologies).

- *Ordering clash:* When two data sets have the same elements but in different orders (e.g., one sorted by name, another by date).
- *Boundary clash:* When grouping boundaries differ between streams. A classic example is the calendar problem: one system might group days into weeks, another into months. Weeks and months do not structurally align; there is always a leftover at month's end that splits a week. Another example from computing is storing records in fixed-size blocks: record boundaries and block boundaries rarely coincide, causing a boundary clash between logical records and physical blocks.

Such clashes mean that a straightforward one-pass program cannot simultaneously step through both structures without some additional mechanism. In JSP, a structure clash is not a failure of the method but a property of the problem. It indicates that the problem inherently requires either multiple passes or intermediate data reorganisation (for example, sorting one file or storing data temporarily). JSP provides techniques for handling each kind of clash. For interleaving and ordering clashes, often the solution is to sort or merge data appropriately before or as part of processing. For boundary clashes, sometimes it means introducing an intermediate file or converting one structure into another. Recognising a clash early in design is valuable because it prevents trying to force an incorrect program structure. Jackson's procedure teaches that if you find yourself unable to make a single coherent structure, you likely have a clash and must resolve it by changing the problem representation (perhaps splitting the job into two programs or introducing a pre-processing step, etc.).

Apart from clashes between data streams, JSP also demands the data be *parsable* in a certain way. Specifically, the method assumes that each input file can be parsed with a one-unit lookahead. This means that by looking at the next record (or a peek at the next input element), the program can determine what branch of the structure to follow. In parsing terms, the grammar of the input must be LL(1) or at least deterministic with a single token of lookahead. If this is not the case, the designer encounters a recognition difficulty. For example, if two different record types in a file share a similar prefix such that you cannot tell them apart until reading further, that would violate the one-lookahead rule. In JSP, recognition difficulties might be resolved by more sophisticated parsing techniques (multiple lookahead, backing up, etc.), but those lie outside basic JSP and verge into general parsing. Jackson's advice was that if you cannot parse with one lookahead, you will struggle to write the loop conditions and if-conditions for the program. You might need to adjust the data format or apply a workaround.

Jackson noted that when such difficulties (structure clashes or recognition issues) arise, they signal inherent complexity in the problem rather than a flaw in JSP per se. JSP's basic method works elegantly when data structures align nicely; when they do not, it simply reveals that the problem will be complex no matter what, requiring either a refined method or additional steps. In the historical development of JSP, these insights led to advanced techniques (like *program inversion* to handle certain structure mismatches, which we will discuss later).

4. Lay Out the Program Structure and Insert Operations

Assuming the structure is decided and free of unmanageable clashes, the next step is to actually design the program logic around that structure. The JSP literature often describes this as taking the empty structure (just loops and ifs) and then placing operations in the program structure. Each data stream operation (reading a record or writing a record) and each processing step (calculating totals, etc.) has a natural place in the structured program. Jackson gave rules for where to put read/write statements: for instance, one should perform an input READ at the start of processing a new record or group, often with one lookahead record always in a buffer. In our earlier example of customers and orders, this means: read the first customer record to start; then, in the outer loop, at the bottom, read the next customer record to prime the next iteration; similarly, inside the inner loop, after processing an order, read the next order record to prepare for the next iteration, and so on. The correctness of loop conditions (like while (another order)) relies on having that lookahead record available to test if it belongs to the current group or a new group. JSP provides a disciplined approach to writing these conditions. For example, the condition on the inner loop might be expressed abstractly as **while** another order exists in the current customer group, which with a one-record lookahead translates into a check that the next record is an order of the same customer. By systematically placing reads at the boundaries of structure components, JSP helps avoid common errors like reading past the end of a file or missing a record.

All computations or data transformations are then slotted into the appropriate part of the structure. If a calculation pertains to each individual record, it goes inside the innermost loop; if something needs to happen at the end of a group (like outputting a subtotal when a customer's orders are done), it would be placed right after the inner loop finishes (still inside the outer loop, but after the inner loop block). JSP's clear delineation of begin/end of structures makes it easier to know where such code belongs. One advantage of this approach is that it localises the effects of changes. If tomorrow the requirement is to add a new summary output at the end of each customer, the JSP-designed program has a clear spot for end-of-customer-group logic. In a less structured program, one might have to insert a check in the main loop for `if` next customer is different `then` output summary, which is more scattered and prone to mistakes. This exemplifies how JSP leads to maintainable code: changes in data or output structure tend to correspond to adding/modifying code in one coherent block of the program, rather than tangled conditions spread throughout.

5. Refine and Optimise if Necessary

After the structured pseudocode is drafted, the final step is writing it in the target programming language. The clarity of the method does not always translate to the tersest code, but clarity was usually prioritised over brevity. Only after getting a correctly structured solution would one consider optimisations (and JSP devoted some attention to that, emphasising that optimisation should not ruin the structural clarity except in truly necessary cases).

To recap this methodology in an example: Suppose we are designing a program to produce a report of sales per region from a transactions file. The input transactions file is sorted by region and within region by product. The output report groups by region, with subtotals per region. Using JSP, you proceed in the following steps:

1) *Data structures*: Input = sequence of (Region header, sequence of Product transactions). Output = sequence of (Region section: Region name line, sequence of detail lines, region total line). These are similar hierarchies (region grouping).
2) *Program structure*: while (another Region) do: read Region header (from input); print Region header (to output); initialise region total; while (another transaction in this region) do: read transaction; process/accumulate; print detail line; end inner loop; compute region total; print region total line; end outer loop.
3) *Confirm no clashes*: input and output are both grouped by region in the same order, so it aligns fine. One record of input (region header) corresponds to one output action (region heading). Transactions flow through straightforwardly. If the output required something

like sorted by product across regions, that would have been a clash with input sorted by region, requiring perhaps a different approach (such as accumulating and resorting, etc., but in our case, it is aligned).

4) *Place operations*: reads and writes are placed as outlined; calculations (like summing a region total) are done at the appropriate points.

The result is a structured program that naturally handles region boundaries (when region changes, the outer loop triggers ending one group and starting another) and thus does not need extra convoluted logic. This process is essentially building a tailor-made control flow for each specific data format. In contrast to a generic algorithm, JSP yields an algorithm that is hard-*wired* to the data's pattern. This is why the code ends up simpler and often more efficient for that particular task. It is like a custom parser or transducer for the data streams at hand.

Before moving on, it is important to acknowledge that JSP's method works best when the data structures are relatively fixed and known in advance (which is true of e.g. data streams in device drivers are file processing tasks). If data structures are ad hoc or very dynamic, JSP might be less directly applicable. But even then, thinking in terms of "What is the structure of the data we expect?" can clarify and influence program design.

Having seen how JSP helps in designing a program, we now transition to examining the deeper connections of JSP's approach to finite state machines and parsing, as well as discussing the advanced concept of program inversion, which extends JSP to more dynamic scenarios. This will highlight the theoretical underpinnings of JSP and its relevance to automata and state-based thinking.

4.2 JSP and Finite State Machines: Parsing and State Alignment

One of the insightful ways to view JSP is through the lens of formal languages and automata. A JSP-designed program, with its structure derived from a data format, is essentially implementing a parser or transducer for that data format. The program's control flow can be seen as a kind of finite state machine (FSM) or pushdown automaton that recognises the input structure and triggers corresponding processing actions. Consider the structure of input data as a *language* defined by a grammar. In the earlier example, the grammar for the customer order file is

```
(CustRecord (SimpleOrder | UrgentOrder)* )*
```

This grammar can be recognised by a finite state machine with a small amount of memory (in fact, a pushdown automaton if we consider nested structures, or simply a deterministic parser with one-token lookahead, which for regular-like grammars is effectively an FSM with some state for being inside a customer group). The JSP program with its nested loops is effectively a coded version of such an automaton. Each level of the loop corresponds to being in a certain state or context (e.g., we are currently inside a customer group, expecting either an order or the end of the group). The conditions that control the loops and if-statements correspond to transitions in a state machine (e.g., if the next record is an order, stay in this loop; if not, exit the loop and thus end the group). In this sense, JSP was taking what a compiler-writer might think of as a *specification of a language structure* and directly turning it into an implementation.

This connection was not explicitly made by Jackson in the early days, but the relationship between data structures and program structure is essentially the relationship exploited in parsing by recursive descent. Recursive descent parsing is a technique where the structure of the code mirrors the grammar of the language being parsed. Each nonterminal in the grammar corresponds to a function or block of code, and each production rule corresponds to control flow (like choices and loops) in that code. JSP is essentially recursive descent for data file formats, except that it was developed independently in a business programming context. It even dealt with similar issues: for instance, a recognition difficulty in JSP (needing more than one lookahead record) is analogous to a grammar that is not LL(1) and thus not suitable for simple recursive descent without backtracking.

A finite state machine aspect comes strongly into play when we consider how the JSP program transitions between different parts of the input or output. Each possible combination of where we are in each file can be thought of as a state. A simple JSP loop is like an FSM that continues in a state until a condition triggers a transition to a new state (like finishing a group and going back to the top of an outer loop). In fact, one could draw a state transition diagram for a JSP program's execution. States could be [processing a customer; awaiting the next order] or [between customers; about to start a new customer], etc. However, the JSP methodology assures that these states and transitions are systematically derived rather than ad hoc. The state machine is not something we design separately, it is embedded in the structured code. This perspective also helps us appreciate JSP's handling of multiple streams. When a program processes two input streams in parallel (like merging two files), the combined behaviour can be thought of as an FSM that has to handle events from either file in the correct order. A JSP design for merging might treat, say, the next record coming from File A vs from File B as two branches in a conditional, carefully looping to interleave them sorted by key. The method of aligning structures can be seen as synchronising two automata (each reading its own file) into one joint automaton (the program) that produces output. If one file has no equivalent of something from the other, that is where a structure clash or special case arises. Essentially, the automaton for one file cannot be synchronised step-for-step with the other, requiring a different approach.

Another explicit tie between JSP and state machines appears in the concept of program inversion, a technique introduced to deal with certain structural problems. Program inversion involves taking a well-structured program (designed as above) and algorithmically transforming it into a different form that can be called repeatedly and resumed where it left off. This is akin to converting a main program into a subroutine with memory of its state between calls, essentially making it into a coroutine or an iterator. Jackson realised that by doing this, one could extend JSP to handle interactive scenarios. For example, instead of reading an entire file and then outputting results, an inverted program could be called each time an external event occurs, processing incrementally. The states that the program goes through become stored in what JSP calls a state vector, and the inversion introduces a state variable to represent where to resume.

The significance of program inversion is that it generalises the applicability of JSP to even more programming tasks. Jackson and his colleagues realised that many event-driven or interactive processes, which appear to be continuously running or waiting for input, can be thought of as if they were processing a sequential stream of inputs over time. For instance, an interactive dialog with a user can be viewed as a sequence of user input events; an operating system's background process can be seen as processing a stream of interrupts or messages. If you can conceptualise the series of events over time as a data stream, you can design a JSP program to handle that stream as a whole (as if reading from a file of all events). Then, by inverting the program, you obtain a routine that can handle one event at a time but with the guarantee that its internal logic remains correct and complete. This inverted routine is like an event handler that is internally implementing a state machine, and JSP's design ensures that the state machine is built methodically and covers all cases. Indeed, Jackson noted that program inversion was closely related to the concept of coroutines (as in Simula's semi-coroutines) and to the notion of Unix pipes, which let one program's output feed into another's input as a stream. Both coroutines and pipes essentially break a sequential process into interacting pieces while preserving the idea of a sequence of data being processed. Relating this back to finite state machines: any time we have a resumable process, we can model its behaviour as an FSM where each suspension point is a state. JSP, via program inversion, gives a systematic way to derive those states from an initially sequential design. This underscores that JSP, at its core, is about recognising the implicit state transitions dictated by data structures and making them explicit in program structure.

It is also worth connecting JSP's data-aligned structure to the formal notion of *regular expressions* and *regular languages*. The structure diagrams are essentially a visual form of regular expressions (with some extension towards context-freeness for nested structures). A program that follows a Jackson diagram is effectively implementing a deterministic finite automaton (DFA) that matches the regular pattern of the input. The difference between a typical DFA and a JSP program is that the JSP program also performs actions (like computing or outputting) as it recognises the pattern. In parser terminology, it is a parsing with semantic actions. In the automata world, this would be a Moore or Mealy machine (an FSM with outputs) where outputs are triggered by certain transitions or states. When the JSP program writes an output record or updates a total, that can be seen as the output of the state machine in response to certain input patterns. Thus, JSP's approach can be interpreted in terms of finite state machines and parsing:

- The structured program is effectively an automaton that recognises the structure of the input (and ensures the output is produced in a structured way).
- Each level of a loop corresponds to being in a certain context or state in the input.
- Conditions on data (like checking record types) determine transitions (which branch of a conditional to take, whether to exit a loop, etc.).
- The method's constraint of one-record lookahead ensures the automaton can make decisions without ambiguity (a DFA needs a finite lookahead or none for regular languages).
- Program inversion turns a sequential automaton into an event-driven one, introducing an explicit state variable to track where the automaton left off between calls.

Understanding JSP in this way reinforces why it leads to less convoluted code: you are effectively designing a deterministic machine for your problem rather than inventing an arbitrary imperative program from scratch. Deterministic machines, when properly derived, have no extraneous complexity; every state and transition is there for a reason, corresponding to something in the input or output structure. If one does not follow such a method, one might still end up creating an implicit state machine, but in a haphazard way with states encoded in various flags and control flow scattered across the code. That implicit FSM can be very hard to reason about or modify because it was not consciously designed.

In modern practice, explicit state machines are often used for certain tasks (like protocol handling, UI navigation, etc.), but for many everyday coding tasks, the connection to FSMs is not recognised. JSP reminds us that every program that processes structured input is in some sense a state machine recognising that input. By being deliberate about this fact, we can write more systematic programs. The next section will delve into the kinds of problems JSP was designed to solve, particularly focusing on the issues of program clarity and maintenance, and examine why neglecting these principles leads to error-prone and costly code, as we often see today.

4.3 Solving Real-World Problems

JSP was not developed as an academic exercise. It was very much aimed at real-world programming problems that Jackson and others encountered in industry. The key problems JSP set out to solve were unnecessary complexity in code, difficulty of maintenance, and brittleness in the face of requirement changes. By aligning the program structure to the data structure(s), JSP tackled these issues head-on.

One clear problem in many programs (especially those written without a guiding method) is that the code's structure does not make the underlying task obvious. We have all seen (or even written) functions where the control flow jumps around based on flags, special conditions, and partial checks of data, making it hard to see the big picture of what the program is logically doing. In data processing tasks, a common source of such complexity is handling boundaries and special cases. For instance, recall the scenario of counting duplicate records in a file: the naive single-loop implementation had to explicitly handle the first record of a new group and the last record of the file as special cases. This adds extra if statements that are easy to get wrong. One might forget to output the last group's count, for example, a classic bug. JSP's structured solution with nested loops eliminated those particular if statements by making the grouping an inherent part of the loop structure. Thus, it naturally handled the first and last records of each group. The resulting code is clearer since it directly reflects the problem logic of grouping and less error-prone because it is harder to forget a case when the structure drives the logic.

Maintainability is closely related to clarity. A program that is structured in an intuitive way is easier for someone else (or the original author months later) to understand and modify. The JSP philosophy yields maintainability, especially by *localising the effects of changes*. As mentioned earlier, if the data format changes, the corresponding code changes in a JSP-designed program tend to be in one place that mirrors that format part. This reduces the chance of ripple effects causing bugs. It also reduces the cost of implementing changes. Many anecdotal accounts from the JSP era claim that modifications that would have taken days in a non-JSP program (due to needing to trace through complex logic) could be done in hours with a JSP program because the structure guided the implementer to the right spot.

This maintainability aspect has direct economic implications. Studies in software engineering have long noted that maintenance accounts for a large majority of software's total life-cycle cost, often cited as 60–80% or more of the cost. When code is convoluted and lacks a disciplined structure, each maintenance task (fixing a bug or adapting to a new requirement) takes longer and risks introducing new bugs. By contrast, a well-structured program can drastically cut down that effort. In JSP's case, the promise was that if companies adopted this method for their programs, they would save significantly in the long run due to easier enhancements and fewer errors. Indeed, it was reported that many programmers have expressed their experience that JSP gave them insights about program design that they never had before. For the first time, they understood how to design programs they had been writing without really understanding them. This insight often translates into writing it correctly the first time and avoiding the endless debug cycles of trial-and-error coding.

Another practical problem JSP addresses is consistency. In a development team, if everyone follows a method like JSP, the resulting programs have a similar look and structure (provided they solve similar kinds of tasks). This means a programmer can more easily take over another's code because the overall form is familiar. There are even standard ways to draw structure diagrams and standard heuristics for handling common patterns. This consistency can greatly improve team productivity and reduce the learning curve for new team members or for moving developers between projects.

Yet, despite these advantages, JSP and similar methods have faded from widespread use. Why? One reason might be that modern programming environments mask some of the complexity that JSP was dealing with. For example, interactive applications now often use frameworks (like web frameworks or GUI toolkits) where the control flow is managed by the framework and the programmers only write event handlers. In such cases, a developer might not think about designing the overall control structure at all. It is provided by the framework, which internally might be using state machines, but that is hidden. However, even in these environments, neglecting structure can cause problems. Consider a web application where the flow of pages or API calls has some logical structure (say, a multi-step form); if the developer does not design that flow clearly, they might end up with a mess of flags in session state to know where the user is, and bugs if the user navigates out of order. In effect, they have ignored a data structure (the sequence of steps) and thus their code has to handle seemingly weird transitions. A JSP mindset would have led to writing a clearer sequence controller for the multi-step process.

Another reason JSP faded is the advent of object-oriented programming (OOP). OOP encourages structuring code around data in a different way: bundling data with operations (methods) in objects and using inheritance hierarchies to represent categories. While OOP excels at certain kinds of abstraction and code reuse, it does not inherently solve the problem of aligning control flow with data sequences. It is quite possible (and indeed common) to write object-oriented code that still has convoluted logic for processing inputs since the code inside each object is still subject to the same design mistakes. Besides, OOP is outright unsuitable for certain kinds of programming tasks. For example, one might have objects representing different record types but still handle the reading loop in an ad-hoc manner. Some of JSP's advocates have argued that OOP, as taught, often emphasises the static structure of data (class design) and not the dynamic structure of data (how data flows through over time), which JSP addresses. There is no contradiction between OOP and JSP. One could design an object-oriented system whose methods use JSP principles internally. But in practice, few did that explicitly; JSP was viewed as a legacy of procedural programming already by the 1990s, and the focus shifted to design patterns, UML diagrams of classes, etc., rather than a structured flow of control.

The consequence in many modern codebases is that while the architecture might be well thought out at a high level (services, classes, modules), the nitty-gritty of how each piece processes input is left to individual developers' intuition. Some may implicitly or inadvertently follow JSP-like reasoning (thinking carefully about loops and conditions based on input structure), but others may just hack something together that works. Over time, such code accumulates technical debt, e.g., lots of edge-case handling scattered around. The long-term maintenance cost can be huge, as evidenced by industry surveys that up to 80% of software costs are spent on maintenance and evolution. A portion of these costs undoubtedly arises from having to maintain code that was not structured methodically to begin with.

JSP's focus on data streams also resonates with a current trend: the rise of data-centric programming such as stream processing, JSON or XML processing, etc. Ironically, JSP's idea is quite applicable there. For instance, processing a nested JSON would benefit from code structure mirroring the JSON schema. Yet, often programmers either use high-level libraries (like JSON mappers) or write custom code without an explicit method. Those who write custom code sometimes reinvent mini-JSP approaches, like recursively processing JSON arrays and objects in a structured way, essentially because that is naturally how one handles hierarchical data if doing it carefully.

Further, there is also the question of computer science education. Surveying typical CS curricula today, hardly any teach courses in structured programming. Introductory courses often focus on syntax and basic algorithms, while higher-level courses jump to data structures, object orientation, or software engineering with design patterns. The specific needs of a program design method between coding and software architecture are often not addressed. It is assumed that students will somehow pick up good structuring practices implicitly. Some educators incorporate design recipes. For instance, in some functional programming courses, there is a notion of design by data analysis, which is conceptually similar to JSP but in a functional paradigm. By and large, though, structured programming's absence in curricula means new graduates are unlikely to have even heard of it, let alone apply it.

This lack of awareness is why JSP's principle appears obvious yet neglected. An experienced programmer, when shown a JSP solution, might say, "Well, of course the program should look like that." But the reality is that without being explicitly taught, in school or through apprenticeship, not everyone will naturally devise that structure, especially under schedule pressure or when dealing with unfamiliar data. It seems easier to bang out a quick loop with some `if` conditions that seem to work than to step back and design a two-level loop with a clear structure. The payoff of doing the latter comes later during maintenance or extension, whereas the payoff of the quick hack is immediate since the program works for the basic case. Unfortunately, when the code later needs changes, the original developer might be gone or the deadlines tighter, and the next person has to patch the patch, leading to cumulative complexity, one of the least attractive effects of ad-hoc programming.

From a cost perspective, such undisciplined development leads to what is often called technical debt. Each messy solution is a "debt" that will need to be paid (often with a huge interest) when modifications or debugging occur. Structured programming can be seen as a way to avoid incurring certain types of technical debt by investing upfront in a good structure. The cost to design a JSP solution might be slightly higher initially (because it requires careful thinking and maybe drawing a structure diagram), but it saves costly rework and errors later. This is analogous to how structured programming in general was justified: it might take a bit longer to structure your program well, but you greatly reduce debugging time. As an example, a PTS bank application that adopted JSP for its report programs might find that adding a new field to a report is straightforward (just add handling in one place), whereas otherwise, it might have risked breaking the report or having to rewrite big parts. Multiplied over dozens of programs and years, that difference can translate to significant savings and reliability improvements.

To summarise, JSP was designed to solve the problem of *how to make programs simpler and safer to change* by exploiting the inherent structure of data. It addresses the spaghetti code issue not only by banning `GOTO` (which structured programming already did) but also by giving positive guidance on program organisation. The result, when applied, is programs that are easier to read, reason about, and extend, attributes directly tied to lower error rates and cost. Modern coding practice, which often underemphasises explicit methodology, suffers from the loss of such guidance. As a result, we frequently see overly complicated functions that parse input formats (say, log-file analysers, file converters, etc.) with logic that is much harder to follow than necessary. Reintroducing JSP's ideas, or at least its mindset, would mitigate these issues.

In the next and final section, we will discuss the enduring value of JSP's principles and consider how and when they should be applied today, acknowledging that JSP is not a silver bullet for all scenarios but is a highly effective approach for the scenarios it was meant for.

5. Modern Relevance of JSP

Although JSP originated in a very different era of computing, its core ideas remain surprisingly fresh and applicable. In fact, as software projects have only grown larger and more complex, the need for disciplined design at the code level is arguably even greater today. The decline of JSP's popularity does not correspond to a loss of its relevance; rather, it reflects shifts in industry focus and the overshadowing of fine-grained program design by other concerns. By revisiting JSP, we can reclaim some valuable guidance for contemporary development. In this section, we articulate why JSP's foundational principle still matters and discuss when it should (and should not) be used.

5.1 The Unique Contribution of JSP

The most important contribution of JSP is instilling the mindset that *whenever you have clearly structured input or output data, you should let that structure drive your program design*. This sounds obvious once stated: why would you structure a program in a way that fights its input format? Yet, countless codebases demonstrate that this wisdom is often not applied. Developers may ignore an input's hierarchical nature either out of haste or because they have not been trained to think in those terms. By applying a method such as JSP, one essentially reduces design decisions to following the data template. This does not remove creativity from programming; rather, it channels creativity into the right areas (like how to handle the content of each section) instead of reinventing control flow wheels.

Think of scenarios today: reading an XML or JSON document, processing a stream of events such as messages from a queue, generating a report from database results, writing a compiler or interpreter, handling nested configuration files, etc. All these tasks involve data that has an intrinsic structure. A modern developer might address those using libraries or frameworks, but at some level, if they write code for it, they confront the same issues JSP addresses. For example, many programming languages now have built-in support for parsing JSON into objects. But suppose you did not use a direct mapping, instead you might write code to navigate the JSON structure. If you follow a method like JSP, you would likely write a recursive function or loops that mirror the JSON's schema. If you do not, you might try to handle it in one big loop with a lot of `if`/`switch`es on node types, and that can get messy quickly. JSP's advice in such situations is embedded in various best practices. For example, using a recursive approach to traverse a tree is one way of saying follow the data structure. JSP formalised that kind of general sound programming advice into a general method.

When to Use JSP

JSP is not a panacea for every programming problem, and it is important to know when it fits well. It shines when dealing with sequential data processing, i.e. when your program's primary role is to read data (from one or multiple streams), do some processing, and write the data. Here, read and write should be taken very broadly as consuming and producing data, which can be from and to e.g. arrays, not necessarily streams like files or sockets. This covers a huge class of programs, even today. Scripts that parse logs, services that consume message streams, utilities that transform files, components that generate reports or summaries, etc. These all benefit from JSP thinking. If the input or output is in any way hierarchical or ordered, JSP is well suited. On the other hand, if your program's behaviour is not driven by structured data, JSP might not directly apply. For example, algorithmic computations like sorting or computing a numerical result are not about the input data structure in the same sense. It is more about direct operations on data in memory. A good rule of thumb is that if the description of the data access pattern of a procedure is very complex, then JSP is less applicable. Thus, JSP is less directly applicable to purely interactive systems where there is no notion of a pre-described defined sequence of inputs. It is not the interaction per se that is the hindrance. As we discussed earlier, through program inversion, one can often still apply JSP by conceptualising the sequence of events over time. However, if a system is highly concurrent or reactive without a clearly describable input or output pattern, other design approaches, such as event-driven design, might take precedence. But even then, JSP's usefulness might come into play by modelling each concurrent component's behaviour as a pseudo-data sequence.

JSP might not be necessary for trivial cases either. If an input is extremely simple, say just a flat list of numbers to sum, one does not need JSP. A single loop is trivial and there is no hidden structure to worry about. The value of JSP grows with the complexity of the data structures. When there are multiple levels of grouping, multiple input/output files, or various record types, that is when JSP provides a clear advantage by preventing mistakes and clarifying design. JSP is applicable per procedure or set of procedures, so it is not an all-or-none decision for developing a system.

5.2 JSP in Modern Languages

One reason sometimes given for JSP's decline is that modern high-level languages (like Python, Java, C#, etc.) and their ecosystems reduce the need for such methods. A language with powerful libraries can indeed let you handle data without writing as much explicit control flow. For instance, if reading XML, one might use an XPath query to grab pieces without writing loops. But under the hood, something is still doing that structured processing. Moreover, not all tasks can be accomplished by off-the-shelf tools. There is still plenty of custom data handling in applications. Using a modern language in no way obviates the need for structured thinking. It only changes the syntax and the level of application. A smelly goat dressed in satin is still a smelly goat. In fact, JSP can be implemented in any language, be it procedural, object-oriented, or functional. It is about organising code, not about specific APIs. For example, one can implement a JSP design in an object-oriented style by having objects for each structure element and methods that get called in a nested fashion. Or in a functional style by having mutually recursive functions for each level of data grammar. The concept transcends paradigms.

5.3 Combining JSP with Other Paradigms

There is a clear opportunity to integrate JSP principles within contemporary development practices. For instance, in an object-oriented design, one might have a class representing the overall job with a method like `process()`, and inside it, methods that process substructures. That class might use other helper classes representing input sources or output destinations. This does not conflict with JSP; it merely encapsulates the structured program into an object. Similarly, one could use design patterns like Iterator or Visitor to implement a JSP approach. For example, a Visitor pattern over a composite data structure is akin to a structured traversal.

To leverage JSP today, the main hurdle is the mindset. Programmers need to be made aware of the approach and understand when to apply it. This means it could be beneficial to reintroduce these concepts in software engineering education. Even if not taught as JSP by name, the idea of designing programs by first writing down the structure of inputs/outputs (maybe as a data schema or grammar) and then sketching a matching program flow should be taught as a standard practice. In requirements engineering, people often create data models or use case flows. JSP essentially connects a data model to a program design.

5.4 The Cost of Neglecting Structured Programming

We have discussed how modern coding often defaults to intuition or expediency. It is worth emphasising the tangible cost of that. Large systems frequently become expensive to maintain precisely because their lower-level code is full of irregular, patchwork logic. Such code breeds bugs: edge cases missed or inadvertently broken by changes. It also slows down new feature development because developers must untangle what's there to safely extend it. If JSP's principles were widely applied, many modules and components would likely be smaller, more coherent, and easier to adjust. Not every module deals with sequential data, of course, but for those that do, applying JSP could prevent them from becoming the weak links in the system.

One can draw a parallel with test-driven development (TDD) or other agile practices: TDD encourages writing tests first to shape the code's interface and ensure reliability. JSP could be seen as data-driven design that shapes the code's internals for reliability and clarity. They are complementary: one ensures correctness via tests and the other ensures a logical organisation of code. A team that ignores either may initially produce working software, but over time the lack of structure or lack of tests (respectively) can make the software brittle.

5.5 Reintroducing JSP Principles

To make use of JSP in today's organisational programming environments, teams or organisations do not necessarily need to adopt a formal JSP methodology with diagrams and such (though they could). It might be enough to incorporate its principles into code reviews and design guidelines. For example, a PTS guideline is: If your function or module processes input or output data with an identifiable structure, ensure that your control flow reflects that structure. Consider using nested loops or recursive functions corresponding to the data's hierarchy, and avoid flat processing that requires explicit state flags for different cases. This is essentially a restatement of JSP in a modern context. Code reviews can catch violations. For instance, if someone writes a 200-line function that reads a file and uses a bunch of flags to know where they are in a file, a reviewer might suggest refactoring it to a clearer structure by breaking it into sub-functions that handle each section of the file. Another tactic is to make use of grammars and parser generators for complex input formats. This is common in compiler development using tools like ANTLR, Yacc, etc. In a sense, that is an automated application of JSP ideas. You write the grammar (data structure) and a tool generates a parser (program structure) which you then augment with semantic actions (processing logic). Not every problem warrants a full parser generator, but it is an option for complex input structures.

Ultimately, the argument for JSP's enduring value is an argument for *methodical programming*. It is a call to avoid the complacency of just-code-it-now-and-trim-it-later, by injecting a bit of upfront analysis of the data's form. As many systems continue to be maintained for decades, clarity at the code level is not a luxury; it is essential for sustainability. The principle that program structure should follow data structure introduces regularity and predictability which, in turn, reduce errors and facilitate changes, outcomes any software team would desire.

In conclusion, while JSP as a brand has faded, the problems it addresses have not disappeared, and neither has the method lost its efficacy. In an age where software is pervasive and maintenance is a major cost driver, applying JSP principles can help cut down those costs by preventing unnecessary complexity. It is a piece of software engineering wisdom that deserves to be remembered and reapplied in modern practice. Good ideas in programming often resurface. Hopefully, it is time for JSP's ideas to resurface and be given its due recognition in the context of contemporary software craftsmanship.

Appendix: A Note on Jackson Structured Development (JSD)

While this essay has focused on JSP, it is important to distinguish it from Jackson Structured Development (JSD), which Jackson introduced in the 1980s. JSD took the ideas of JSP and applied them to the design of entire systems, particularly systems characterised by a number of independent but interacting processes (like a bank accounting system with multiple transaction types, or a real-time control system). JSD had three phases: modelling, network, and implementation, where one would model entities and their actions over time (each essentially a sequential process, often shown as an entity structure similar to a JSP structure), then consider how those processes communicate (network), and finally implement them possibly via inversion and other techniques. JSD is a broader methodology and is beyond the scope here, but it is worth noting that JSD's *modelling phase* essentially applies JSP's thinking at a higher level: each entity's life history is like a data structure (sequence of events) for which you could write a program. In fact, one could say JSD recognised that even in interactive systems, there are underlying sequential structures, i.e. the sequences of events each entity undergoes. JSD then systematised designing a system from those sequences. However, JSD is less known and was one of many competing system development methodologies in the 1980s. It competed with the likes of SSADM, Yourdon's SA/SD, OMT, etc., and later with the OOP/UML movements. We mention JSD mainly to clarify that when we talk about JSP's relevance today, we are focusing on program design at the code level (the design of an algorithm or module handling data), not full system architecture. JSD would be relevant if one were designing, say, a whole complex event-driven system with Jackson's approach, but that is a separate topic. Here we limit ourselves to the programmatic set of principles that JSP championed: *structure the program like the data*.