Does Beautiful Code Matter? We think, So...

by

Nicholas Vaidyanathan

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved September 2018 by the
Graduate Supervisory Committee:

Dr James Collofello, Chair
Dr. Robert Atkinson
Dr. Hessam Sarjoughan
Dr. Hasan Davulcu

ARIZONA STATE UNIVERSITY

December 2018

ABSTRACT

In this dissertation, I explored the state of the literature of software engineering and found that while multiple schools of thought exist, most practitioners argue there is no consensus on a theory to guide software engineering [1, 2, 3]. Agile practitioners have developed a set of guiding principles and tools commonly advocated in industry such as the SOLID design principles for writing code, but these remain experiential rather than experimentally verified.

Cognitive Science developed a theoretical framework for guiding the development of curricular material called Cognitive Load Theory (CLT)[4, 5]. CLT derives from the earliest discoveries of neuroscience [6], its central precepts involve the design of instructional content and arrangement in a way that is cognitively available. CLT seeks to understand the human memory model, specifically the limits of short-term "working memory" [7], and optimize content for memorability. CLT has been effectively applied in a variety of instructional materials and verified via lab experiments since the in 1970s.

Software is essentially a concretion of a programmer's understanding of the world to create emergent behavior. The core activity of programming is the organization and arrangement of information as realized in data structures and algorithms. Software has explored the complexity of this practice through a variety of metrics, some of which include McCabe's Complexity Metrics [8], Halstead's Software Science [9], Albrecht's Function Points [10], and Wang's Cognitive Complexity Metrics [11, 12]. I found no examples of known metrics that leveraged CLT in their development.

My work explores a conceptual link between CLT and software engineering best practices. I provide a partial mapping of refactoring techniques and SOLID principles to CLT principles. This link moves software engineering towards a theoretical framework based on human cognition. This dualistic tie can help both fields. Instruc-

tional Designers can organize webs of content according to distributed systems design principles, while software engineers can leverage principles backed by a theoretical framework, experimental approach, and known principles of human cognition.

I designed an experiment that explored the effects of applying these principles on an established software library. I measured the perceived cognitive load, time to debug/mean-time to resolution, and defects introduced via broken tests using a 2x2 Factorial Design with experienced and novice software engineers. With a sample size of n=188, I measured that the average mean time to resolution and number of bugs reported by both experienced and novice programmers. I found that the mean time to resolution and introduced defect rate is less for both experienced and novice developers when debugging the refactored code, aligning with concepts from Cognitive Load Theory. I also find that those programmers reported less perceived cognitive load.

Agile design principles combined with the precepts of Cognitive Load Theory can produce software that is measurably easier to debug and understand. Augmenting known refactoring patterns with additional heuristics–such as managing the size of classes and methods to around Miller's Magic Number and naming concepts according to their usage– produces novel software architectural principles as a consequence of this work. This has implications on Cognitive Load as a measurement for and conceptual backbone of technical debt. Future work should probe advanced ways of measuring cognitive load and programmer experience and the effect of programming language and domain.

This research provides a significant contribution by applying concepts and experimental design from CLT to software engineering. The study measurably shows that code refactored according to specific principles designed to manage cognitive load results in software that is better understood and easier to debug. Looking at

code comprehensibility for programmers through the lens of CLT may lead to new techniques of quantifying readability and analyzing technical debt.

To my Poobah, who has never stopped believing in me and always inspired me. I love you.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

Software engineering is one of the youngest and most dynamic fields in engineering. Naur and Randell helped formally define and introduce Software engineering as an area of study in 1968 with their landmark NATO software engineering report. [13] The most striking finding in the report was the startling conclusion that 68% of all software projects fail. Since then, researchers have investigated many different approaches to extract software from the "tar pit" [14]. Explorations have spanned a variety of both technical and non-technical solutions, spawning research in areas as disparate as software cost and estimation, software architecture and design, and software project and process management. Nevertheless, despite the dedicated effort of researchers and practitioners, in over 40 years since the landmark initial paper, research has made minimal ingress against the problem. A follow-up report by the Standish group in the 1990s suggested that overruns are still close to 189% [15], though the numbers are in dispute [16, 17]. The 2015 CHAOS report shows that 52% of projects are challenged, 19% failed [15].

One possible root cause of this churn arises from the inherent difficulty of the base activity of software engineering: computer programming. Leading software engineering practitioners argue that there are better ways to engineer software [18, 19], the Agile software philosophy. Tools such as Test-Driven Development [20, 21], Design Patterns [22, 23], and Object-Oriented Programming [24, 25, 26, 27]. Unfortunately, there is a dearth of empirical evidence supporting these practices [28] [1]. Moreover, software engineering has yet to develop a strong, unifying theory that can serve as an conceptual framework for software engineering study [2] [3].

In parallel, cognitive science developed from the frontiers of psychology into a broad, impactful field that enhanced understanding of the human brain. Developing quickly from George Armitage Miller's insight about the limits of human working memory [6], cognitive science blossomed into a myriad of disciplines exploring different aspects of human cognition. One of these fields emerged in the 1980s when John Sweller studied problem solving and developed Cognitive Load Theory (CLT) . Based on the work of Miller, Baddeley's working memory model [7], and collected evidence about short-term "working" memory, Cognitive Load Theory studies effects the arrangement and presentation of learning content has on learner memory and performance. From years of development across a variety of domains, CLT has shown evidence for particular phenomena such as the Split-Attention Effect [29] and the Redundancy Effect [5]. Instructional Designers adopted the findings of CLT into their materials and began to achieve measurable, repeatable results that validated these findings and produced more efficient learning material.

These two fields are evolving independently, in parallel. But should they? Fred Brooks describes software as complex to build because "The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures..." [30] Additionally, Kent Beck describes the purpose of programming as "What is it that we want to get out of code? The most important thing is learning. The way I learn is to have a thought, then test it out to see if it is a good thought. Code is the best way I know of to do this." [31] Elements of memory, organization of information, and learning appear prodigiously through software literature, but ties to CLT are scant to nonexistent.

The question of whether CLT could inform the theory and practice of software

engineering and its fundamental activity of computer programming drove this research. In this dissertation, I showed that it can. I suggested that a sampling of the best practices of object oriented analysis and design corresponded to conceptual best practices in Cognitive Load Theory. Using this premise, I designed an experiment similar to traditional CLT experiments that measured the efficiency of programmers to debug and fix software as a function of time to resolve an issue, defects introduced, and perceived cognitive load. I found that amongst both novice and experienced programmers, more people were capable of fixing a bug in refactored code, and that it took less time and introduced less average defects. I found that the subject's internal measurement of Cognitive Load in the refactored code was less than the original code. This suggests that CLT can make a promising contribution into the scientific understanding of the complexity of software systems, providing a basis to evaluate software designs and understand technical debt.

Chapter 2

LITERATURE REVIEW

In the introduction, I gave a high level overview of software engineering and cognitive load theory and the origins of the fields. In this section, I start by reviewing the literature of software engineering and computer science as it has related to specific attempts to influence the practice of programming. Software engineering research covers a space as disparate as processes surrounding the elicitation of software requirements, visualization techniques for software designs, estimation practices for predicting delivery cycles, probabilistic testing, and more. I viewed CLT specifically from the frame of how it can influence the practice and understanding of programming. I progress chronologically, looking at specific programming recommendations of the 1970s and 1980s and moving towards the 2000s and 2010. I also review the way computer science has attempted to quantify the concept of complexity in computer programs through the development of metrics. I find that although CLT is neither directly invoked nor indirectly referenced, much of the modern practitioner literature of Agile Software Development and Software Craftsmanship [32] extensively focus on the comprehensibility of computer programs. The practices and principles proposed in these works have conceptual similarity to cognitive load theory. By the end of this section, the reader should have a sense of established research that is similar to mine in software engineering and a high-level overview of cognitive load theory.

Others have attempted to improve programming practice in the past. Edsgar Djikstra published the provocative *Go To Statement Considered Harmful* [33] in a letter to the editor of Communications of the ACM in 1968. This paper, premised with the assertion that the "the quality of programmers is a decreasing function of

the density of go to statements in the programs they produce" essentially helped launch the conversation that "the way you write code matters" by arguing that one particular tool in high level programming languages, the goto statement, made programs harder to understand. Djikstra continued refining these ideas and published a set of notes which led to a book on Structured Programming [34]. Structured Programming prescribed the use of control structures– such as if/else statements, cases, and looping statements–instead of only variable assignment statements and jumps to labeled sections. Structured Programming influenced the development of new programming languages like Kernighan and Ritchie's C Programming Language [35], which incorporated control structures as the fundamental primitives of writing code. Structured programming triggered the elaboration of programming paradigms as diverse as imperative programming, procedural programming, and object-oriented programming. Imperative programming simply arranges program statements in order of execution, very similar to writing a task list. Procedural programming organizes imperative code into subroutines, named groups of imperative statements that can form a basis for abstraction and apply Parnas principle of information hiding [36]. Object-oriented programming defines systems in terms of message passing between independent units that encapsulate data and the operations on that data called objects [37]. In a parallel track, Donald Knuth developed Literate Programming [38, 39]. Programmers who write in a "literate" style combine prose and narrative to elaborate the function of the application. The author of a literate program writes with readability and comprehensibility in mind as a direct consequence of the formulations in accompanying text. Literate Programming has yet to penetrate the majority of the programming zeitgeist, remaining a fairly niche research interest. Nevertheless, many authors have experimented with it and reported enhanced clarity [40]. There are many other paradigms of programming, as have been extensively collected and

classified by Van Roy [67]. All of these paradigms seek to apply a specific set of constraints and models of interaction to computer programs to make it easier for a programmer to reason about and implement their design. Some researchers have looked at the potential cognitive elements of programming paradigms [41, **?**], others have looked at cognitive elements independently of programming paradigm [42], but the discipline lacks a de facto standard for the evaluation of the cognitive impact of programming paradigm.

Rather than measure complexity through the impact on human cognition directly, Software metrics comprise effort by the software engineering research community to quantify the results of the software process. The original, most cited software metric is the Line of Code (LOC) and the analogous KLOC (thousands of lines of code). An oft-maligned metric— that does not account for the internal complexity or structure of a program— LOC has nonetheless remained common due to simplicity of measurement. Attempts to provide more information about the inner working of a program have led to the creation of metrics such as Halstead's Software Science [9], McCabe's Cyclomatic Complexity [8], and Albrecht's Function Points [10]. These approaches have found limited success. Empirical studies have produced very little conclusive evidence as to the efficacy of these metrics [43], with many equivocal examples [44]. Modern metrics research, however, is attempting to incorporate elements of cognitive science into the art of understanding software. Wang's study of the cognitive complexity of software serves as a bridge for the metrics work into this world [11] [12]. Wang develops a set of meta-cognitive models for the description and understanding of computer programs, then constructs a set of experiments that compare and contrast the cognitive complexity of different constructs. The argument is that the metrics generated through such analysis are more accurate descriptors of software complexity than symbolic, structural, or functional complexity metrics.

Cognitive metrics for software scratches the surface of what cognitive science can offer computer science and software engineering. Software maintainers spend more time reading code than they do writing new code. Many developers who write new code often find themselves working with unfamiliar technologies or in an unfamiliar domain. People commonly believe there are similarities between programming and education, but no unifying link in theory has tied successful programming outcomes to established educational theory. One such relevant theory is Cognitive Load Theory [45], which informs the way educational designers seek to develop curricular materials. Human Computer Interface (HCI) designers leverage it to provide better user experiences [46, 47]. Nevertheless, programmers have not applied this work towards actually changing the way they write code for other programmers. Software engineering research has been focused on other areas.

Software Engineering literature of the early 1990s continued the work of the 1980s in software processes and modeling, leading to the development of the Unified Modeling Language (UML) [48]] and the associated Rational Unified Process (RUP) [49]. In 1994, the Gang of Four published their seminal Design Patterns [22] catalog, a hybrid design/architecture and implementation publication that inspired a rush of patterns oriented research activity in the community, including implementation patterns [50], testing patterns [20], enterprise application architecture patterns [51], language implementation patterns [52], and others. In February 2001, a group of experienced and well-respected software engineers published the Agile Manifesto [19]. A counterpoint to RUP, agile software development emphasized less documentation, tools, and repeatable processes. Instead, it championed greater emphasis on relationship building and collaboration. This approach quickly became popular with industry, especially with the rise of web application development. Its emphasis on results effectively captured the spirit of the rise of the Internet and fast-paced nature of web development.

Consequently, an explosion of interest for effective ways of organizing and managing agile software teams led to the development of models such as Extreme Programming [31, 53], and Kanban [54].

Many of the forerunners of Agile were leading evolution in multiple areas of software engineering. Andrew Hunt and Dave Thomas wrote The Pragmatic Programmer [55], fleshing out the baseline principles of the Agile Manifesto into a programming philosophy built on pragmatism. Kent Beck helped elaborate Extreme Programming [31], codified best practice Implementation Patterns [50], and wrote the most popular Java variant of the xUnit testing framework, JUnit, as part of his work in proselytizing Test-Driven Development [20]. Martin Fowler developed a widely regarded Refactoring patterns catalog [56] extending from Bill Opdike's PhD thesis [57] to provide a structured, behavior-preserving methodology for modifying existing code to promote clarity and reuse. Robert C. Martin helped expound some basic principles that have become the bedrock of SOLID object-oriented software development [58], including the Single Responsibility Principle, the Interface Segregation Principle, and Dependency Inversion alongside Barbara Liskov's Substitution Principle [59] and Bertrand Meyer's Open/Closed Principle [60].

These industrial contributions have ironically transformed much of the software engineering landscape—particularly in web and mobile software development, the preeminent paradigms of the 21st century—at arguably a broader scale than academic software engineering research. Recognition of this dichotomy between the research emphasis and industrial literature has fueled the rise of the Software Craftsmanship movement [32]. Software Craftsmanship espouses treating programming as a craft more than a science or an engineering discipline. In such epistemology, dedicated artisans construct the best software. Such artisans hone their craft through discipline, practice, and apprenticeship/journeyman style interactions with masters.

One could see Software Craftsmanship as a reaction against and rejection of Software Engineering. It eschews the emphasis on process, models, and up-front work to build software systems. Alternately, one could argue that it represents a competing school of thought. Much like Cubism and Expressionism were competing schools of thought in 20th century painting, software craftsmanship and SWEBOK [61] [56] based approaches may be seen as competing schools of thought within 21st century software engineering. One can relate this to the difference shown in DeRemer and Kron's "Programming-in-the large versus programming-in-the-small" [62]. Many software craftsmen seek to apply cognitive science to improve their craft. Andy Hunt wrote a book to help programmers "refactor their wetware" by applying concepts from cognitive science [63]. They have yet to link cognitive science to writing code. CLT is the link.

2.1 Establishing the link between Cognitive Load Theory and Software Engineering through Software Craftsmanship

Before I dive into the conceptual overlap between software craftsmanship best practices and CLT, it will be helpful to define the concepts individually. I begin by outlining concepts from software craftsmanship, then defining concepts from CLT, then describing their connection.

The Single Responsibility Principle states that "a class should have only one reason to change." [58] For example, the design of a vending machine class may include methods to add money, vend a product, set the prices of individual products, restock individual products, and refund money. If the currency of the machine has to change, the implementation may need to be modified. If the product stocking needs to support batches, the implementation may need to be modified. This design conflates inventory management with price calculation. A Single Responsibility Principle-based

design would break apart the functionality such that the vending machine has an inventory and a cost calculator. Application of the SRP tends to break larger objects into smaller, more cohesive objects. This increased cohesion is reminiscent of the cognitive concepts of chunking, essentially the SRP calls for smaller, independent chunks. It also aligns with CLT guidelines of paring content down to essentials and writing high coherent text for low-knowledge learners.

2.1.1 Open-Closed Principle

The Open-Closed Principle states that software entities should be open for extension but closed for modification [58]. This seemingly unachievable goal drives designs towards simple abstractions that change rarely but can add behaviors to a system through different implementations. For example, an online retailer may select the right hello message to display on a landing page based on the country the customer is in based on a Greeter class. If that class uses a hardcoded array, whenever that retailer expands to a new country, the Greeter will have to change. If instead the design employs a STRATEGY for Greeting, new implementations can be added without modifying old code. This also optimizes chunking to differentiate between abstraction and implementation, which is reminiscent of the CLT guidelines of teaching system components before teaching the whole process and teaching supporting knowledge separate from procedure steps.

2.1.2 Newspaper Metaphor

The Newspaper Metaphor is described as a way of arranging code like the headlines and sections of a newspaper, with headlines and headings that describe the content to follow [18]. In this style of arrangement, a programmer tries to co-locate functions with their usage such that the narrative flows from abstract to concrete. For example,

"In order to parse HTML, one reads from a string and tokenizes it. In order to read from a string, one reads character by character into a buffer until reaching the null terminator. In order to tokenize, one splits the string by a delimiter..." This most directly relates to CLT in terms of sequencing. It is reminiscent of the CLT guideline of giving learners control over pacing and managing cognitive load when pacing must be instructionally controlled.

2.1.3 Design Patterns

Design Patterns emerged from Software Engineering adopting the approach of Christopher Alexander [64] to describe different architectural patterns and applying it to software. There are patterns for object-oriented analysis and design, implementation, software architectures, and many other areas of software engineering. Design Patterns evince the CLT guideline of imposing Germane Cognitive Load to enhance efficiency by helping learners automate new knowledge and skills, forming a specialized lexicon that can enable experienced and proficient practitioners to communicate complex concepts quickly.

2.1.4 Intrinsic, Germane, and Extraneous Cognitive Load

Cognitive Load Theory often classifies load into three types: Intrinsic, germane, and extraneous [5]. Intrinsic Cognitive Load is the basis of CLT, the irreducible complexity of concepts. Intrinsic Cognitive Load derives as a function of the number of distinct elements working memory needs to process and the interconnections between them [7]. CLT does not seek to eliminate intrinsic cognitive load as it cannot be eliminated without obliterating the underlying concept. Instead, CLT seeks to manage intrinsic cognitive load. Two strategies often employed to limit the intrinsic cognitive load are sequencing and chunking [4]. Chunking takes similar elements and

groups them together, allowing the working memory to treat the numerous units as a single unit for storage and retrieval. Sequencing sorts the elements in an ordering that enables progression with minimal backtracking or cross-referencing. Germane Cognitive Load is the cognitive load involved in activities that is actually in service to the desired outcome. For example, doing different types of problems when learning equations to see how they apply in various contexts can increase the germane cognitive load. Learning specific vocabularies such as Design Patterns to succinctly communicate concepts also increases Germane Cognitive Load. Extraneous cognitive load is caused by a sub-optimal arrangement or presentation of content demands more working memory attention from things that are irrelevant. Redundant verbiage or distracting pictures can introduce extraneous cognitive load by shifting attention away from the desired goal. A lot of CLT work seeks to manage intrinsic load, optimize germane load, and reduce extraneous load as much as possible.

2.1.5 Split Attention Effect

Colloquially, split attention occurs when one is reading a passage and feels annoyance when it instructs them to flip back or forward for more information [65]. It very similar to the idea of the overhead involved in context switching when it comes to concurrent thread management in software engineering. The Split-Attention Effect is mitigated by integrating content to be self-descriptive and cohesive without having to jump back and forth.

2.1.6 Expertise Reversal Effect

An important finding for instructional design is that many of the techniques effective to help novices acquire schemas have no effect or actually impede the understanding of experts [66]. As someone gains experience and builds the necessary

mental schemas, it's often helpful to reduce the cognitive load management methods used because they serve as substitutes for pre-built schemas for novices. In any study examining CLT with respect to software, it's helpful to consider if evidence of the expertise reversal effect can be observed. For instance, LIterate Programming when writing a compiler may be extremely helpful for someone unfamiliar with compiler theory and design, but may actually impede the understanding of an expert who is just trying to find the language's grammar and doesn't need a lesson in Context Free Grammars in the comments.

2.1.7 Applied Example of the types of Cognitive Load in software

As an applied but simplistic example, consider the case of an electronic retailer. Perhaps in an initial rapid prototype the retailer's website offers some static Hyper-Text Markup Language to greet a customer.

```
<html>
   <head>
      <title>An E-commerce retailer</title>
   </head>
   <body>
      <h1> Hello Dear Customer!</h1>
      <h2> Please <a href="buy/12345">buy this product</h2>
   </body>
</html>
```

Figure 2.1: insert caption

The Intrinsic Cognitive Load of this markup includes an understanding of the syntax and semantics of HTML, the ability to read the English language, and the knowledge that there is another component referenced by a hyperlink that completes the purchasing process. The Germane Cognitive Load includes some semantic understanding of the intent of the application–such as knowing that the 12345 corresponds to a product identifier that is used by another page to trigger purchasing. The Extra-

neous Cognitive Load includes the lack of an ending close tag for the anchor- -. Someone reading this document later may have difficulty understanding whether the intent was to only make the word "buy" a hyperlink, or whether the intention was to "buy this product."

Suppose market research discovers that more purchases occur if the greeting is warm and friendly. The product team asks the development team to change the greeting to a formulation that has been shown to lead to more purchases, addressing the customer as "Dear, dear customer." The development team has design options. They may implement the change by simply changing the text, as shown below:

```
<html>
   <head>
      <title>An E-commerce retailer</title>
   </head>
   <body>
      <!-- Feature1: Market research revealed
      that "Dear dear Customer" leads to more sales -->
      <h1> Hello Dear dear Customer!</h1>
      <h2> Please <a href="buy/12345">buy this product</h2>
   </body>
</html>
```

Figure 2.2: insert caption

This simple change adds another word to the document, but has minimal effect on the intrinsic, germane, or extraneous cognitive load. A comment has been added to clarify why this changed.

Suppose further market research reveals that "Dear dear Customer" works well for people who shop with their browser set to the Spanish language, but "Valued Shopper" works better for others. The Intrinsic Cognitive Load of this application will increase. Either different pages will need to be loaded by the HTTP server based upon the customer's location, or the markup will need to be updated in a way that

enables the greeting to be displayed dynamically. The development team may choose the second option and generate the code below:

```
<html>
   <head>
      <title>An E-commerce retailer</title>
   </head>
   <body>
     <!-- Feature1: Market research revealed
     that "Dear dear Customer" leads to more sales -->
      <h1> Hello <span id="greeting">Placeholder</span>!</h1>
      <h2> Please <a href="buy/12345">buy</a> this product</h2>
   </body>
   <script type="text/javascript">
      var loadGreeting = function() {
        var span = document.querySelector("#greeting");
        if ('textContent' in span) {
          span.textContent = greeting();
        } else {
          span.innerText = greeting();
        }
      }

      /*
      used for language detection from
      https://stackoverflow.com/questions/1043339/javascript-for-detecting-browser-language-preference
      */
      var greeting = function() {
        var lang = window.navigator.languages[0] ||
        [window.navigator.language || window.navigator.userLanguage]
        if (lang === "es") {
            return "Dear dear Customer";
        }
            return "Valued Shopper";
      }

      document.onready = loadGreeting();
   </script>
</html>
```

Figure 2.3: insert caption

Not only has the Intrinsic Cognitive Load of this application increased conceptually, but the Cognitive Load imposed by the code has increased dramatically as well. A dynamic element has been introduced to the document through the use of Javascript, which will replace the HTML with an id of "greeting" rendered text based on the detected value of the language when the document is ready. The Germane Cognitive Load now includes an understanding of Javascript language syntax, semantics,

and eventing model, localization issues of language, and the platform-specific nuances of different browsers. The comment that was helpful in the static "simple" update is now Extraneous Cognitive Load, because what is rendered is no longer just "Dear dear Customer", it may also be "Valued Shopper." Future developers may question whether the code is correct because the comment tells them it should do one thing, but the code itself does another.

The Intrinsic Cognitive Load of the application cannot be reduced without changing its functionality. This Cognitive Load can be managed by sequencing and chunking. An enterprising developer may refactor the application as show below:

```
<html>
  <head>
    <title>An E-commerce retailer</title>
  </head>
  <body>
    <h1> Hello <span id="greeting">Placeholder</span>!</h1>
    <h2> Please <a href="buy/12345">buy</a> this product</h2>
  </body>
  <script src="greeting.js" type="module"></script>
  <script type="module">
    import greetCustomer from './greeting.js'
    greetCustomer(document.querySelector("#greeting"));
  </script>
</html>
```

Figure 2.4: insert caption

```
import getUserPreferredLanguage from './language.js'
import crossPlatformSpanTextUpdateCommand from './cross_platform_span_update.js'

export default function greetCustomer(location) {
  loadGreeting(location, getUserPreferredLanguage());
}

function loadGreeting(span, lang) {
  var text = greeting(lang);
  crossPlatformSpanTextUpdate(span, text);
}

function greeting(lang) {
  if (lang === "es") {
      return "Dear dear Customer";
  }
      return "Valued Shopper";
}
```

Figure 2.5: insert caption

```
/*
used for language detection from
https://stackoverflow.com/questions/1043339/javascript-for-detecting-browser-language-preference
*/
export default function getUserPreferredLanguage() {
  return window.navigator.languages[0] ||
  [window.navigator.language || window.navigator.userLanguage];
}
```

Figure 2.6: insert caption

```
export default function crossPlatformSpanTextUpdateCommand(span, text) {
  if ('textContent' in span) {
    span.textContent = text;
  } else {
    span.innerText = text;
  }
}
```

Figure 2.7: insert caption

This refactoring introduces the concepts of sequencing and chunking. It adds Germane Cognitive Load by using JavaScript's modules functionality, but the complexity of the original HTML file is dramatically reduced as its bits are chunked across multiple files, meant to be read and understood sequentially. I propose the following hypotheses as links between software craftsmanship practices and CLT:

- The Single Responsibility Principle is a technique that reduces Extraneous Cognitive Load and manages the Intrinsic Cognitive Load of classes by promoting smaller classes. In the above example, the single responsibility of the HTML page is to invoke the greetCustomer function.

- The Open/Closed Principle helps beginners manage Germane Cognitive Load by enabling consumers of classes to leverage existing components while avoiding cognitive overload by abstracting away inner details. In the above example, getUserPreferredLanguage is a default function whose implementation is closed to modification, but communicates intention through its name.

- The Newspaper Metaphor is a technique for minimizing the Split Attention Effect, reducing Extraneous Cognitive Load. In the above example, methods are arranged according to their usage.

- Design Patterns form a higher-level vocabulary that increases Germane Cognitive Load for intermediate/advanced developers, allowing richer conversations with less distracting detail. In the above example, the COMMAND pattern is used to convey that a function has a void return type and performs a side-effect.

- Smaller functions/methods with descriptive names align closely with Miller's Magic Number Seven Plus or Minus Two [58], allowing programmers to hold more of a method's functionality in their heads at once. This reduces context switching and makes the overall component easier to understand. This pattern scales. Small methods, small classes, small packages, small libraries, small frameworks are more cognitively available for novices, enabling quicker, less error-prone development. In the above example, none of the functions have more than 7 lines.

2.2 Theory & Predictions

Cognitive Load Theory has shown measurable and repeatable results that have influenced Instructional Design. Refactoring patterns are widely accepted for reducing technical debt and making code easier to understand and modify, but it is hard to quantify this effect. In some software engineering companies, the process of peer code review is used as a mechanism for quality control and to manage technical debt. In code review discussions, the application of certain Refactoring techniques or the order in which they are applied can give rise to disagreements that they are simply about style versus substance. Someone without previous background in patterns literature may argue that the introduction of a pattern makes code more complicated. Sometimes one with such a background may argue that the wrong pattern is being applied. Decision points about whether a block of code should become a separate method or class, the names of identifiers, the control structures leveraged, and other points of the software design and implementation can become a horse trading affair where software engineers try to persuade each other based on appeals to authority, tribal knowledge, and individual preference. Cognitive Load Theory offers a structured approach for measuring and quantifying the technical debt of a software project, and a conceptual framework to guide these discussions. Software teams can apply a practice of periodically reviewing certain classes in their code-base and subjectively reporting the estimated cognitive load. Such a practice would allow, over time, to use the measured change in reported cognitive load as a metric to include in the calculation of the technical debt of a project.

Additionally, CLT guidelines can change the way we think about documenting and understanding code. For instance, an application of the Modality Effect suggests that embedding UML diagrams directly in source code that involves spatial reason-

ing, and providing audio narration could improve the comprehensibility of the code. Integrated Developer Environments such as Eclipse Juno, IntelliJ IDEA 2014 and Visual Studio 2010 don't support this type of experience natively. The guidelines of worked examples and self-explanations can provide a cognitive basis for the efficacy of Test/Behavior Driven Development. Guidelines related to separating system components from processes and supporting knowledge from procedure steps can inform arguments behind the viability of Interface Oriented Design [67] and the choices behind particular abstractions.

The complexity of software most often becomes an issue when the code has to change. The primary drivers of change in software are finding a defect or desiring a new feature. When the code must be changed, there is a risk that previous correct behavior may become incorrect, or that the performance profile–in terms of CPU utilization, memory usage, IO, or other factors– may change in a dramatic way that affects the availability of the system. This is when the technical debt associated with the system may cause the necessary change to take hours, weeks, or more. If Cognitive Load Theory can quantify the technical debt a system experiences effectively, one would expect to find that systems with high technical debt impose high cognitive load on their maintainers and users. These high technical debt systems would contain more design flaws, more implementation bugs, and changes to these systems would take longer to implement and validate, hence being more likely to introduce regressions from desired behavior. Conversely, systems with lower technical debt should impose lower cognitive load. The number of design flaws and implementation defects should be lower, the average time to resolve issues should be less, and changes should introduce fewer regressions. Consequently, code that is refactored to manage its cognitive load more effectively should be code where one can more easily fix an outstanding bug, the attempt at fixing should introduce less regressions on average,

and the engineers who attempt the fix should report less perceived cognitive load from the code they read and modify. A useful experiment would look at an existing piece of software with well-defined functionality and a known bug, create a refactored version that maintains the presence of the bug but more effectively manages the cognitive load, then verify that it is easier to fix the refactored version and that engineers feel less cognitive load while trying.

2.3 Research Question: Can Cognitive Load Theory provide empirical evidence and a conceptual framework for the efficacy of Refactoring?

The non-directional null-hypotheses to test, in mathematical terms, are:

1. Time of completion (control) = Time of completion (refactored)
2. Perceived Cognitive Load (control) = Perceived Cognitive Load (refactored)
3. Average defects introduced (control) = average defects introduced (refactored)

These hypotheses should be testable with a 95% ($p < .05$) confidence level and desired power of 80% (π=0.80).

2.3.1 Why will this be meaningful?

One would expect to find the code refactored to manage cognitive load requires less average time to fix defects and introduces less regressions while trying. This will enable the research community to explore further interventions and evaluate questions such as "do some principles have a higher impact when applied than others?", "is the effect of the principles in any way tied to programming language?", and "could results be affected familiarity with program domain?" Additionally, showing the efficacy of Cognitive Load Theory's application to software engineering creates a wealth of opportunities to explore its recommendations for visualizations, organization of project

documentation, and personalized tooling support for software engineer of various skill levels.

2.3.2 What could go wrong?

It is possible that authors wildly misrepresent the effects of the programming heuristics. Consequently, divergent results upon measurement may be unlike previous Cognitive Load experiments. Perhaps even with classical techniques of measuring Cognitive Load one will be unable to find a relationship between programming performance and Cognitive Load optimization techniques. This would suggest CLT is not as strong a fit for software as the original hypotheses suppose. Additionally, Cognitive Load Theory states the principle of the Expertise Reversal Effect, which says that some practices that highly benefit novices will flummox experts and vice-versa. A robust experiment should identify instances of this by targeting an equal mix of novice and expert participants. One would expect that content increases Germane Cognitive Load, such as the lexicon of Design Patterns, would improve the performance of intermediate and proficient practitioners but may induce cognitive overload—and hence have a negative impact—on novice performance. Alternately, partitioning methods and classes to minimize the risk of cognitive overload may be unnecessary for an expert. Such efforts may improve novice performance, but inhibit the analysis of an expert who can "hold more of the code in their head."

2.3.3 How can one measure the Cognitive Load of code?

The Cognitive Load Theory literature describes multiple different options for measuring cognitive load. Paas et al provide a useful literature review [68] that surveys analytical and empirical methods such as subjective rating, physiological response, and concurrent-task assessment. In separate work, Paas et al compared subjective

ratings with psychophysiological response readings and found that subjective rating is reliable and sensitive enough to be useful without the significant cost and implementation barriers that come with more technological measurement [69]. Brunken included direct forms of cognitive load measurement such as EEG [70] and found that a dual-task method may work well for multimedia. Given that the introduction and measurement of CLT are new when it comes to software, I chose a subjective rating scale where participants numerically rank items based on perceived difficulty– commonly known as a 7-point Likert Scale [93]– to measure the induced cognitive load of code snippets.

With these conditions in mind, the next section presents the experiment that tested these hypotheses.

Chapter 3

OVERVIEW OF PRESENT EXPERIMENT

In this section I detail why certain participants are selected, which piece of software is used for the trial and why it is picked, the experimental procedure and how it relates to the research questions driving this work. At the end of this section, the reader should have the necessary background to understand the experiment.

3.1 Participants

In this study, I sought to achieve meaningful results with a 95% confidence interval and 80% power looking at the differences between real production code with refactoring supported by cognitive load theory principles applied and code without. I wanted to explore the difference in mean time to fix a bug, regressions introduced, and perceived cognitive load between experienced programmers and novices. Using the G*Power application [73], I computed a minimal sample size of 185 participants. For ease of calculation, I got 188.

The development practice of those who have multiple years of experience is different from those without, as is commonly seen in expert/novice studies in areas such as chess [71], circuit analysis[72], and nursing [73]. Studies in programming have explored expert schema generation and ways of reading code [74, 44, 75]. Many of these studies have very basic modeling of programming expertise, classifying participants as experts or novices. Models such as the five stage Dreyfus Model [76] have not been widely applied in the research literature. Nevertheless, companies often hire programmers with criterion based on years of experience, despite the lack of a strong research body of evidence that shows correlation between expertise and years pro-

gramming. For the purposes of this study, I chose to segment years of programming experience based on common industry qualifications: <5 years for novices, 5+ years for experienced.

3.2 Materials

3.2.1 Software Chosen - Joda Time

For this study, I used the Joda Time date/time library. Joda is a popular open-source library used in thousands of projects for date/time manipulation for Java. Java already has a Date/Calendar library built into the language designed for these types of operations. However, many developers within the Java community were dissatisfied with the complexity of the interface Date/Calendar provides for achieving operations such as Date/Time arithmetic, time zone conversion, and formatting dates to standard formats such as ISO8601. Because of this complexity–which this research can show is a measurable difference in the cognitive load imposed by these two libraries– Joda Time has become so popular that the author of the library was asked to completely re-write date/time handling for Java 8.

3.2.2 Why Joda Time?

1. Joda Time seeks to reduce the complexity of an existing interface, meaning that it has made an effort to manage the intrinsic cognitive load of date/time manipulation for users.

2. Joda Time has a very large user base and is a high impact project within the Java community. Joda Time sits in the dependency tree of many popular software packages such as Spring, the Lift web application framework, Hibernate annotations, and many, many more. As such, this is not a contrived exercise

dreamed up by a graduate student in a lab to show an effect. This is real code, battle-tested and used by millions.

3. Joda Time is written in Java. Java and C++ are the most well-known programming languages in the community. Java is taught in many college curriculums; finding familiar programmers is less difficult than other languages.

4. Joda Time solves a very general problem–date & time manipulation. Most languages have date libraries for common operations like arithmetic, formatting, or timezone conversion. Good and usable libraries provide broad, usable abstractions for complicated functionality. Date & Time manipulation is very complicated, often causing bugs even in major professional software engineering environments. Programmers across different software engineering domains—whether they build web applications, defense contracting products, or video games—are likely to be able to "grok" it. While esoteric details of handling locale differences and nanoseconds may themselves be complex, the generality of the problem makes it more suitable for development by engineers with various backgrounds than a task such as modifying a Machine Learning library like Weka.

3.2.3 What experimental intervention did I make to Joda-time?

I used two versions of Joda Time. The control version is unmodified source from GitHub, forked from the master branch on July 24th, 2015. The experimental version applies Refactoring to the control version aligning with precepts of Cognitive Load Theory such that:

- Variable identifiers are renamed for clarity

  - Clean Code: Avoid mental mappings → CLT: Integrate Explanatory Text Close to Related Visuals on Pages and Screens to Avoid Split Attention

- Each method has no more than 7+-2 lines
  - Clean Code : functions do no more than one thing → CLT: Write High Coherent Texts for Low Knowledge Readers

- Methods are arranged according to their usage
  - Clean Code : Newspaper Metaphor → CLT : Display Worked Examples and Completion Problems in Ways That Minimize Extraneous Cognitive Load

- Each class has no more than 7+-2 methods
  - Clean Code : Classes do no more than one thing → CLT: Write High

3.2.4 What change did the participants have to make?

Joda Time at this commit contained a bug in its parsing logic for ISO8601 dates. The ISO8601 standard is common enough to be implemented in many different programming languages and be a standard convention for transmitting dates in distributed systems. This makes the bug broadly comprehensible, have mass applicability, and a good candidate for investigation.

3.2.5 Bug: ISO8601 Years

The "devil in the detail" is all of the "edge" cases where bugs happen. One of the devilish details is the existence of a large, amorphous, ever-changing set of formats describing a date. The International Standards Organization attempted to ameliorate this with ISO8601[82], which defines a canonical format for the interchange of dates

and times across locales. One of the quirks of the standard is that the published format is YYYY-MM-DD, however, the standard allows for more than 4-digit years in cases where sender and receiver have agreed upon extra digits by prepending with a + or - [±YYYYY]. Joda Time did not originally account for this. This led to the filing of the bug report:

Figure 3.1: insert caption

Hari Shankar explored the depths of the code, identified an issue, and made a fix. He did not report how long it took to find the bug, nor how difficult it was to come up with a solution. When I first found this issue, it was still open. In October of 2015, Steven Colebourne (primary author/maintainer of the library) merged in Shankar's fix.

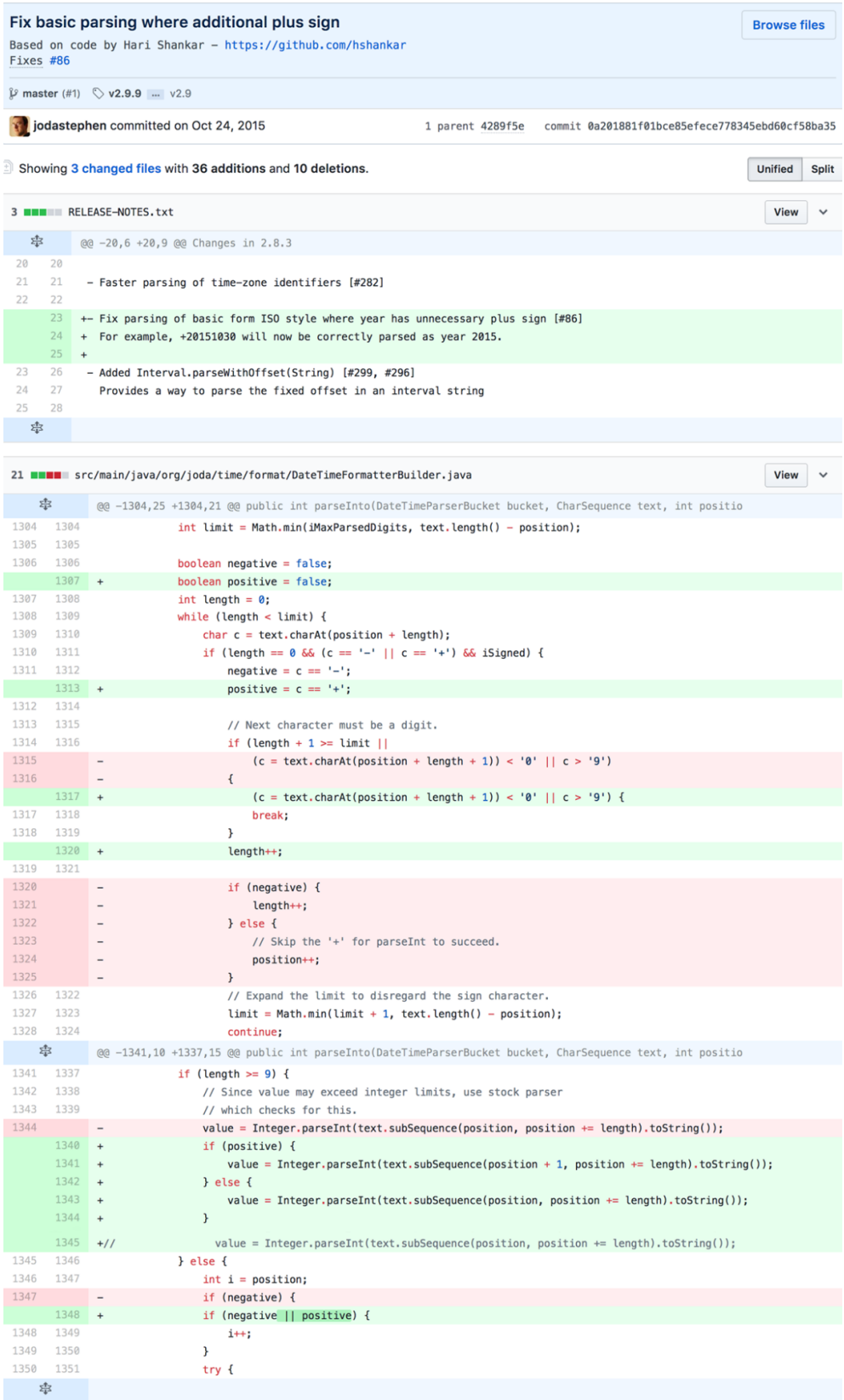


Figure 3.2: Shankar's Fix

3.2.6 Analysis of Accepted Solution

The accepted solution follows a common maintenance programmer practice: change as little code as possible and duplicate where necessary and with slight tweaks to achieve desired behavior. It is not exactly but similar in vein to "Programming By Difference." It is a conservative approach that tends to be favored when code is in obsolescence, has a large number of consumers, or is otherwise hard to change.

"Other reasons" that code can be hard to change include a lack of confidence in correct behavior, or difficulty in understanding algorithms and data structures written by someone else. The commit that introduced the fix included unit test cases that manifest the bug (though notably only try 5 digit years, perhaps not fully exercising failure modes), suggesting the maintainer understood the code enough to augment the existing test suite, a software best practice. The original code was not written solely by Stephen Colebourne. Attribution information in the comments suggests it was a collaborative effort between Colebourne, Brian S. O'Neill, and Fredrik Borgh. It's possible this had an effect on the aggressiveness of the fix.

Using SonarQube 5.4 for analysis, I found that the `DateTimeFormatterBuilder` alone had 1600 LoC. 2625 total lines (including comments) with 162 issues found via static source analysis, with 2.3% duplications, estimated to take 2.5 days to fix. It had a Cyclomatic Complexity score of 550, the highest in the library.

When gathering statistics of the library with the "Run Tests with Coverage" option of IDEA IntelliJ, Joda Time exhibits an impressing 98% class/91% line/90% method coverage. Such a high level of coverage potentially provides developers the option to experiment with more drastic structural Refactoring and verify existing behavior, but time and other issues may not always make such re-architecture possible.

For the purposes of this experiment, I hold the state of JodaTime as reflected in

Git commit on August 2nd 2015 as the "control" group, not including the given fix. I use the provided fix as the "gold standard", as it was accepted by the core library contributor. I then ask participants to attempt to fix the bug following a short tutorial on the ISO8601 standard and the Joda Time library. I follow a 2x2 factorial design where I account for novice/expert and control/experimental measuring debugging time, number of defects introduced as detected by failing unit tests, and perceived cognitive load as reported on the Likert Scale. Non-software related factors that potentially introduce error in measurement include time of day, environmental factors in the deviation from "normal programming place" (office, coffee shop, home, et cetera) versus "clean room", and participant mood/focus/mental factors. I treat non-software related factors as nuisance variables.

3.3 Experimental Environment

To run the study, I developed a lab machine with multiple Integrated Development Environments to suit the needs of Java programmers: Eclipse, IDEA IntelliJ, Oracle NetBeans, Sublime Text Editor, Vim, and Emacs. The lab machine included all necessary software, an Ubuntu 11.10 base OS running Java Development Kit 8 update 21, Git, installed versions of Eclipse Neon, IntelliJ IDEA 14, NetBeans 8.2, Sublime Text 3, VIM 7.1, and emacs 24.2.

This was a conscious concession to permit the introduction of another nuisance variable—IDE familiarity—into the project.

3.3.1 Aside: The IDE effect

Java developers have a lot of flexibility in the toolchain they use, limiting to one particular environment could artificially narrow the participant pool or influence the results. For example, an experienced Eclipse developer may have a significantly

impaired code navigability and debugging experience in Vim, or an Emacs guru could spend valuable code investigation time instead learning the graphical user interface of IntelliJ. Making one tool mandatory could significantly deviate the sample from the programmer population, discouraging some from involvement and hamstringing the rest. I wanted to simulate the field experience of programming as closely as possible, giving participants leverage to integrate prior knowledge and work with code in their naturally preferred workflows. The trade-off is that I did observe effects where toolchain differences had an effect on the study data, including load/processing times and perspective switches of Eclipse and a stability issue with NetBeans.

The in-person approach worked effectively to pilot the study and normalize the workflow and data collection. The only problem was the time-intensiveness of the process. Since the target number of participants was 185, I needed a way of scaling out the study such that participants could independently and in isolation do the exercise in parallel. Consequently, I replicated the lab machine setup in a virtual machine image using Oracle VirtualBox 4.0. I uploaded this virtual machine image onto my website, then wrote a script with instructions sent via email to participants who registered using the online questionnaire. The virtual machine was configured to automatically record the screen and capture audio, creating a very similar experience to the in-person session. Participants would securely send me the captured video file and the filled out Likert Scale for the code listing, along with any notes that corresponded to their impressions after a debrief prompt. This development enabled my research to reach for more participants than it may otherwise have; it seems an ideal way to make a study like this capable of reaching a large number of programmers from different backgrounds all over the world.

3.3.2 Cognitive Load measurement - Likert Scale

I give participants an annotated listing (included in appendices) that applies a 7-point Likert Scale to the code listing in order to analyze perceived Cognitive Load according to common techniques.

Aside: Likert Scale construction

Likert Scales are not commonly applied to code listings. Figuring out how to do so is a point of experimental design. It is an expansive design space, abundant with options. Naively, one may consider applying a scale at points such as every statement/expression, every line of code, every function/method, every class definition, at package level, or even at the bundled executable/library level. I'll discuss some of the strengths and shortcomings of this approach.

Likert Scale: every expression/statement

A Likert Scale on every single expression or statement offers perhaps the most granular sensor for how difficult code is to understand. It's also the most "cost prohibitive" in measurement. Any code whose full functionality that does more computation/evaluation than simple algorithms such as sum/max/min can become incredibly difficult both to instrument this way and for participants to respond. For my study, I chose "real-life production code" precisely to avoid generalized inference of programmer thought process from "toy examples", a common critique of research into the cognitive aspects of software. As measured by SonarQube, the `DateTimeFormatterBuilder` alone has 2625 total lines, and multiple expressions or statements can be in a single line. Giving participants 5000 elements to rank assuredly biases them towards Respondent Fatigue. As this is cutting-edge research intended

to establish the viability of this approach, it is absolutely imperative to minimize corruption of the data based on Respondent Fatigue. I can ameliorate this concern by setting some axioms of things that are "easy to understand"—where one would expect the average participant to respond 1 on a 1-7 least difficult/most difficult scale.

3.3.3 Minimal Cognitive Load of expressions axioms

- Declaration of a variable (e.g. `int x`;)
- Assignment of a value to a variable (e.g. `int x = 2`;)
- Unary and Binary mathematical operations (e.g. `x++`, `x + y`, `x mod y`)

These axioms are challengeable. For instance, the usage of a type in a variable declaration is programming language dependent. There may be measurable differences in simplicity of understanding declarations based on the inherent type systems of the language. The way that such affects comprehensibility may be unexpected. Perhaps in a typeless language such as Assembly, or a language that allow anonymous types/"Duck Typing" such as Javascript, var x = 4 causes respondents less Cognitive Load than int x = 4. Conversely, perhaps the Germane Cognitive Load induced by the assumptions of types such as "an Integer in Java must be a whole number between -231 to 231-1" makes the second statement easier to understand. Both of these conditions may actually be determined by the target programmer's prior experience with programming and languages.

For another example, bit shifting of whole numeric values can be succinctly described as unary/binary mathematics operations, but that may underestimate the cognitive complexity a programmer has to compute to understand what is happening. This comes up in my study in a very concrete way, as the computation of year values during the parsing step contains expressions that befuddled many participants.

Figure 3.3: insert caption

This computation relies on a traditional programmer idiom from the assembly days, that i*10 = (i<<3) + (i<<1) This shortcut is not documented in the code itself and is not one high-level application developers and other software engineers encounter on a daily basis. Every participant that encountered the line during in-person trials reported feeling uncomfortable with it. Some took the time to prove to themselves that the formula worked, many took it as given. Surely this line is not equivalent in cognitive complexity to a simple variable assignment. In these cases, I can compute a metric using Halstead's software metrics or Mira's Modified Cognitive Complexity Measure [77], but none of these metrics could quite capture the Cognitive Load of a prior knowledge lookup from a well-known programmer vocabulary pattern-matching schema. In principle, capturing the subjective Cognitive Load experienced by programmers encountering this line in production code could inform researchers more about the average conceptual vocabulary of programming than other mechanisms.

3.3.4 Likert Scale: line-by-line

A Likert Scale on every single line of code is slightly less expansive than expression/statement, but not by much. Respondent Fatigue is still an overriding concern. The axioms can be expanded further to include line specific concepts.

3.3.5 Minimal Cognitive Load of lines axioms

- Return statements (e.g. `return x + y;`)

- Comments (e.g. `// Next character must be a digit`)

3.3.6 Likert Scale: blocks/scopes

This was ultimately the approach I chose to have participants respond to. Functions/Methods, classes, packages—all of these artifacts are essentially programming language conveniences to define scope at different levels of abstraction. This scope management maps very cleanly to Cognitive Load Theory's concepts of Chunking and Sequencing. Code blocks form lexical and conceptual closures where programmers can partition their understanding of the global state of the program to management pieces. As there fewer than 100 blocks to be analyzed, I avoid the problem of Responder Fatigue but can still capture valuable granular data.

3.3.7 Minimal cognitive load of blocks/scopes axioms

- Getters (e.g. `public int getCurrentPosition() { return currentPosition};`)
- Setters (e.g. `public void setPosition(final int position) { this.position = position};`)

3.3.8 Open Question: How do we measure code flow cognitive complexity?

Likert Scales at the block level get me closer, but still don't quite capture all of the complexity programmers encounter when understanding code. For an applied example, I can examine the code flow of the `DateTimeFormatter`. The `DateTimeFormatter` uses an `InternalParser`,which is implemented by a static abstract class `NumberFormatter` inside of `DateTimeFormatterBuilder`. This `NumberFormatter` is a nested class that serves as the root of an inheritance hierarchy, for classes that include `UnpaddedNumber`, `PaddedNumber`, and `FixedNumber`. Many participants who provided feedback about

the design in the study expressed confusion and incredulity, some expressed outright contempt as they struggled to navigate between classes and trace control flow. This type of activity is simple to observe but difficult to capture quantitatively, and the self-reported Cognitive Load of block of code may not be able to tell us about the cognitive complexity added by interactions between the blocks. This remains an active area of potential innovation in study design and measurement for subsequent research. As a general guideline, software designers should avoid deep inheritance hierarchies or long method chains where the number of jumps in control flow begins to approach human short-term memory limits.

The full Likert scale instrument for measuring cognitive load can be found in the appendices, here I'll show an example question for maximum clarity:

The instrument presents some code–a method of a class– with a 7 point scale where respondents can rank its complexity. For consistency and simplicity of data analysis, in my study this same scale is repeated for every question, only the code changes.

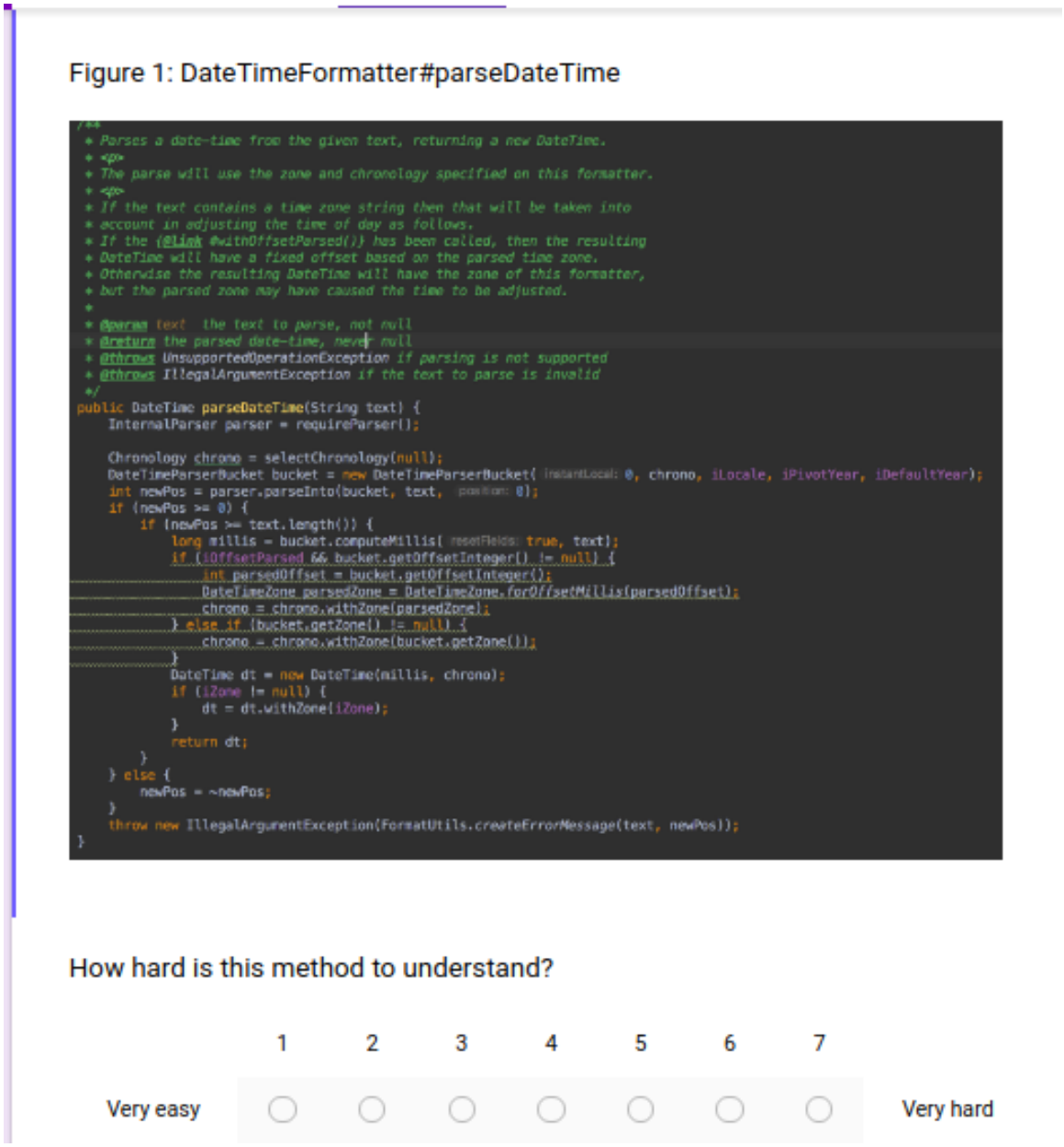


Figure 3.4: Date Time Formatter

In this section I have detailed the rationales for criterion I used to select participants, the software library I used and how I modified it for the experimental purpose, the environment developed to conduct the experiment, and the construction of the measurement instrument for cognitive load. In the next section, I'll detail the execu-

tion of the study.

Chapter 4

METHOD

In the previous section I established that I provided an overview of the experiment I would run to explore a link between software refactoring and its corresponding concepts in Cognitive Load Theory. I proposed this through a classical CLT experimental design: run a 2x2 Factorial experiment split between novices and experienced software engineers for a control and refactored version of the JodaTime Java library trying to fix a bug and measuring time to complete, regressions introduced, and perceived cognitive load. In this section, I detail how I did so. Participants

I solicited participants through an online questionnaire. This questionnaire (included in the appendices) was simply for intake screening and validated eligibility per compliance with the IRB requirements of the study and prerequisite knowledge. This questionnaire was distributed through college professor classrooms, company mailing lists, developer bulletin-boards, and online advertising. I contacted qualifying participants after assigning them to 1 of 4 blocks based on their responses (Novice/Control, Novice/Experimental, Expert/Control, Expert/Experimental), assigning them a randomized identifier, and discarding non-essential personal information to obtain their informed consent. I then scheduled a 1 hour session for conducting the study in person for local participants and sent a link to a virtual machine lab environment for remote participants.

4.1 Materials Design

I developed the experimental refactored version of Joda Time relying heavily on Refactoring patterns while applying the CLT principles of sequencing, chunking, re-

moving redundancy, and introducing germane cognitive load through the direct usage of Design Patterns. The exact same behavior was maintained, while new classes were added using SPROUT CLASS, new methods were added using EXTRACT METHOD, and variables were renamed and re-ordered for clarity. Afterwards, I ran SonarQube 5.4's code analysis for metrics to compare their complexity using traditional software engineering tools. From a pure complexity metrics perspective, `DateTimeFormatterBuilder` went from a control complexity score of 550 to 531 in the refactored. The complexity per function dropped from 3.5 from to 3.4. The number of lines of code dropped by 58. The number of duplications dropped from 2.3% to 0.7%. `DateTimeFormatter` showed a slightly more pronounced effect. The complexity score dropped from 98 to 64. The complexity per function went from 2.5 to 1.7. The number of lines of code dropped by 110. The 2.9% duplications dropped to 0. I've included the full elaboration of the set of transformations in the appendices.

4.2 Intervention, Measures, and Procedure

Participants reviewed a short tutorial on ISO8601 and debugging in their preferred development environment before commencing the timed study. Participants had the opportunity to ask clarifying questions and take notes before they started exploring the code. When ready, they began a 1-hour timed session where they could explore and modify code while trying to fix the bug. When they ran the tests to verify a fix, I took note of failing tests as a measure of regressions introduced. When the participants either fixed the bug or an hour passed, the session ended and I recorded the total elapsed time to the nearest minute. After the session was complete, participants received a survey via e-mail with 7-point Likert Scales with questions where they answered "how hard is this code to understand? (1 = very easy, 7 = very hard)" for the code they investigated.

Chapter 5

RESULTS

5.1 Mean Response Time for participants who fixed the bug

I ran a 2x2 factorial ANOVA using SPSS for the response time amongst successful participants and recorded the following results:

ANOVA for Mean Response Time of fixing bug

Between-Subjects Factors

		Value Label	N
Trial	0	Refactor	27
	1	Control	13
Expertise	0	Expert	27
	1	Novice	13

Descriptive Statistics

Dependent Variable: Time

Trial	Expertise	Mean	Std. Deviation	N
Refactor	Expert	46.42	8.739	19
	Novice	50.50	5.732	8
	Total	47.63	8.082	27
Control	Expert	53.87	6.034	8
	Novice	54.20	6.458	5
	Total	54.00	5.930	13
Total	Expert	48.63	8.643	27
	Novice	51.92	6.048	13
	Total	49.70	7.969	40

Table 5.1: Mean Response Time of fixing bug

Tests of Between-Subjects Effects

Dependent Variable: Time

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared	Noncent. Parameter	Observed Power[b]
Corrected Model	450.093[a]	3	150.031	2.666	.062	.182	7.997	.602
Intercept	83606.728	1	83606.728	1485.383	.000	.976	1485.383	1.000
Trial	247.518	1	247.518	4.397	.043	.109	4.397	.532
Expertise	38.586	1	38.586	.686	.413	.019	.686	.127
Trial * Expertise	28.037	1	28.037	.498	.485	.014	.498	.106
Error	2026.307	36	56.286					
Total	101280.000	40						
Corrected Total	2476.400	39						

a. R Squared = .182 (Adjusted R Squared = .114)

b. Computed using alpha =

Table 5.2: Test of Between Subjects Effects

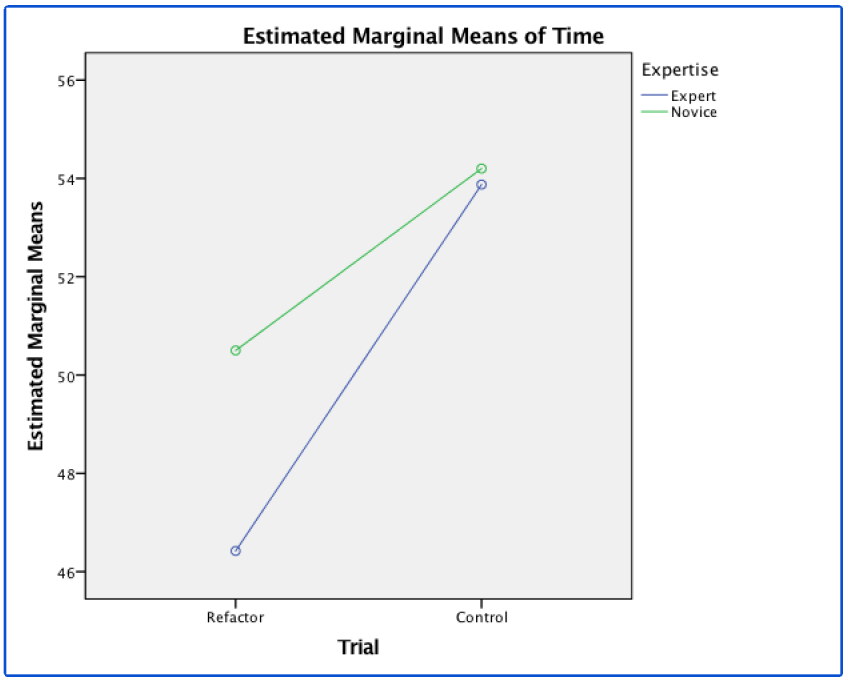


Figure 5.1: Estimated Marginal Means of Time

The data shows strongly that the main effect of Refactoring according to CLT principles causes a statistically significant difference in mean time to resolution for fixing a bug at α=.05. Hence we reject the null hypothesis that the mean time to resolution is equal between the cases.

5.2 Mean Regressions for those who did not fix the bug

I ran a 2x2 factorial ANOVA using SPSS for the regression rate amongst participants who did not fix the bug and recorded the following results:

Regression rate ANOVA for those who did not fix bug

[DataSet1]

Between-Subjects Factors

		Value Label	N
Expertise	0	Expert	64
	1	Novice	81
Trial	0	Refactor	65
	1	Control	80

Descriptive Statistics

Dependent Variable: Regressions

Expertise	Trial	Mean	Std. Deviation	N
Expert	Refactor	42.62	103.499	26
	Control	492.39	972.431	38
	Total	309.67	780.505	64
Novice	Refactor	407.21	1224.063	39
	Control	1002.05	1579.214	42
	Total	715.64	1441.972	81
Total	Refactor	261.37	962.403	65
	Control	759.96	1342.681	80
	Total	536.46	1209.378	145

Table 5.3: Mean Response Time of fixing bug

Tests of Between-Subjects Effects

Dependent Variable: Regressions

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared	Noncent. Parameter	Observed Power[b]
Corrected Model	16170714[a]	3	5390238.15	3.909	.010	.077	11.726	.819
Intercept	33093130.9	1	33093130.9	23.997	.000	.145	23.997	.998
Expertise	6691012.62	1	6691012.62	4.852	.029	.033	4.852	.590
Trial	9553139.04	1	9553139.04	6.927	.009	.047	6.927	.743
Expertise * Trial	184222.229	1	184222.229	.134	.715	.001	.134	.065
Error	194442959	141	1379028.08					
Total	252342376	145						
Corrected Total	210613674	144						

a. R Squared = .077 (Adjusted R Squared = .057)

b. Computed using alpha =

Table 5.4: Test of Between Subjects Effects

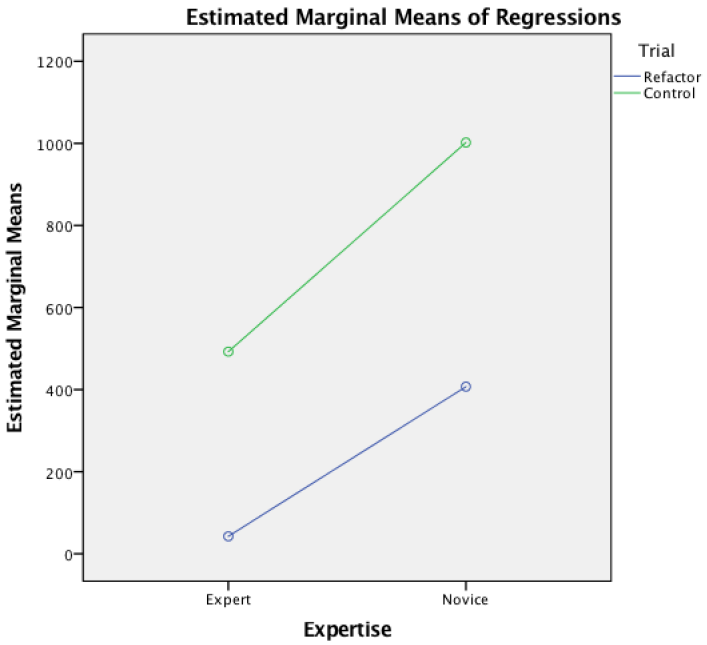


Figure 5.2: Estimated Marginal Means of Regressions

From this data we can observe that there are statistically significant main effects for both expertise and the effect of refactoring, with no interaction effect. Thus, we can reject the null hypothesis that the regression rate is equal between the control code and the refactored code.

5.3 Perceived Cognitive Load

I analyzed the perceived cognitive load by comparing the mean, median, and mode across the control and refactored conditions and examining the differences between overlapping questions. For the refactored condition, the average perceived cognitive load was 3.38, the median was 3, and the mode was 2. Amongst novices, the mean was 3.68, the median was 4, and the mode was 2. Amongst experienced, the mean was 3.08, the median was 3, and the mode was 2. For the control condition, the average perceived cognitive load was 3.73, the median was 3, and the mode was 3. Amongst

novices, the mean was 4.06, the median was 3.5, and the mode was 3. Amongst experienced, the mean was 3.39, the median was 3, and the mode was 1.

The number of questions asked in the survey differed between the treatments, which I'll return to in limitations. There was encountered code that remained unchanged between the two treatments, so exploring the reported perceived cognitive load between them may shed light on the total load imposed by the code base. If the average perceived load is similar, that suggests that programmers partition off the effects of the whole architecture and analyze methods individually. If it is not, that raises the question of whether the perceived cognitive load is influenced by the code around it. For the questions with overlap, the median and mode perceived cognitive load were the same in both the experimental and the control conditions. This suggests that the perceived cognitive load of methods can be reliably measured independent of surrounding context.

5.4 Discusion

i. Time of completion (control) = Time of completion (experiment)

ii. Perceived Cognitive Load (control) = Perceived Cognitive Load (experiment)

iii. Average defects introduced (control) = average defects introduced (experiment)

Based on the preceding results, I can summarize that experience matters in solving a complex problem, as more experts fixed the bug than novices. It required them less time and fewer mistakes on average. But the effect of managing Cognitive Load in code via Refactoring enhances the performance of both. The time of completion and average defects introduced are not equal @ $p < .05$. The average and median of the perceived cognitive load across the 16 experimental questions and the 10 control questions are similar, but the experienced cognitive load in the experimental condition

is often smaller. The control case had 11 questions, the experimental case had 20 (the refactored code was broken out into smaller chunks, so took more blocks to encompass). Some of the code in the questions between the two was the same, much was different. The average cognitive load reported in the code that was the same was approximately equal. This coincides with the theory that Refactoring manages Germane Cognitive Load and removes Extraneous Cognitive Load but does not reduce Intrinsic Cognitive Load. That is usually done by re-engineering. I did not find evidence of the Expertise Reversal Effect when reducing method and class size to more granularly partition out functionality amongst methods and classes. I did not find evidence that the use of Design Patterns terminology applied Germane Cognitive Load to the problem solving process–some experienced engineers self-reported not knowing what a STRATEGY is for the DateTimeParsingStrategy, others felt it was misapplied in this context. More work is needed to identify programmer familiarity with Design Patterns and identify their effect on cognitive load as a function of familiarity.

Chapter 6

LIMITATIONS

For the purposes of selecting programmer populations and sorting by experience, I used a conventional industry classification of years of experience [78] as barometer of expertise. This is widely panned in popular literature [79] and not very well established in the research literature. Many studies are completed with self-selected experts [80] or use distinctions between number of courses taken and skill level [81]. After this study began, new tools that gamify programming such as HackerRank and CodeFights emerged that provide more a more granular analysis of expertise than years of experience. Such tools, if they measure skill and expertise more distinctly, may be better to use in subsequent studies to target experts and novices. Without using such tools, it would be helpful to calibrate participants with 1-2 practice problems before conducting the study and having a panel of experts review their responses to more accurately assign them.

The survey used to measure the perceived cognitive load contained the same language between the experimental and control conditions, but not the same number of questions. It is impossible to rule out that the longer refactored survey had results impacted by respondent fatigue. Although its mean cognitive load was still computed to be lower than the control, in future studies the number of questions should be held constant across both.

Chapter 7

IMPLICATIONS

Finding that the main effect was statistically significant for the Refactoring according to CLT principles but expertise was not for mean time to resolution of the bug was provocative. It suggests that fears that the Expertise Reversal Effect may make the code easier to read only for novices are as yet unsubstantiated. Finding that the perceived cognitive load of overlapping code snippets was similar for both the control and refactored case suggests that CLT may be reliable measure for the technical debt of a software project. Reliability and validity are two important attributes of useful conceptual frameworks and robust metrics, cognitive load as it relates to code may have both to offer the software engineering community. Showing that there is a statistically significant effect for refactored code that adheres to the principles of Cognitive Load Theory introduces new opportunities for the application of CLT to software engineering tooling. Taking into account the Modality Effect when developing documentation to describe a system or constructing tooling to create systems may dramatically affect the way we think about software visualization and implementation. Cognitive Load Theory itself may be informed by software engineering concepts for designing modules/interconnected systems. The principles of Information Hiding, Encapsulation, and resource independence inherent in Service Oriented Architectures may impact the way Instructional Designers organize blocks of content.

Chapter 8

FUTURE DIRECTIONS

The measurement of Cognitive Load in the code was collected via a follow-up survey with Likert Scale on the blocks of code that were actively debugged and engaged. There is a possibility that the perceived ex post facto differs from the load experienced at the time. It would be useful to devise an IDE plugin or some kind of instrument that could get participant feedback in real-time. There may also be interesting results derived through the use of some biometric technologies such as eye tracking, EEG, or fMRI.

This study focused as much as possible on testing the validity of its hypotheses with "real world code." The JodaTime library is at times arcane, complex, and has years of accrued complexity that many participants remarked on throughout the study. There are multiple ways it would be helpful to reproduce these results with different libraries. For one, there may be so much accumulated cognitive load in a large library that looking for smaller libraries with defined problem spaces like JavaPoet may be helpful. Attempting on general purpose code that works in different problem domains such as the Apache Tomcat web application server or the OpenJDK may serve as validation for generality or find specific domains where managing cognitive load produces novel results.

Finally, it's often argued that programming languages themselves have differences in "expressivity" [82] in a way that suggests the intrinsic cognitive load may vary between languages such as Java, LISP, and Python. It would be worthwhile to investigate whether the same techniques yield similar results across languages, or whether the effects are different and language-dependent. Moreover, many of the techniques

we applied were harvested from Refactoring catalogs that had discovered analogues to Cognitive Load Theory based from the Object-Oriented Programming paradigm. It's often argued by practitioners that Functional Programming is more cognitively available, though the pattern literature for FP was not as voluminous when the study began. It would be helpful to have a deeper investigation of the links between functional programming and Cognitive Load Theory.

The measurement of Cognitive Load in code remains an open area of exploration. In this study, I chose to measure based on blocks of cohesive code at the method/function level. It is an open question how to quantify and explore the cognitive load imposed by an entire class as "something more than the sum of the cognitive load of its methods." The relationship between classes, their usage, and how that affects the aggregated Cognitive Load of a component is also worth exploring.

Finally, as systems move from a co-located in the same address space model towards distributed systems/service-oriented architectures, additional complexity is introduced with respect to consistency, availability, and partition tolerance [83]. This complexity is often addressed through tactics that include retrying in case of network failures, caching, and consensus-based algorithms, all of which impose additional cognitive load. As software becomes increasingly complex and responsible for more of the important work of humanity–such as autonomous vehicles and intelligent personal assistants– understanding development practices to make software easy to understand and fix defects in becomes ever more vitally important. Cognitive Load Theory offers a useful set of principles that have already been applied in instructional design, this work suggests it can be useful in software engineering and may help us "escape the tar pit."

References

[1] L. Briand, "Embracing the Engineering Side of Software Engineering," *IEEE Software*, vol. 29, no. 4, p. 96, Jul. 2012. [Online]. Available: http://dx.doi.org/10.1109/ms.2012.86

[2] P. Johnson, M. Ekstedt, and I. Jacobson, "Where's the Theory for Software Engineering?" *IEEE Software*, vol. 29, no. 5, p. 96, Sep. 2012. [Online]. Available: http://dx.doi.org/10.1109/ms.2012.127

[3] I. Jacobson and I. Spence, "Why We Need a Theory for Software Engineering," Oct. 2009. [Online]. Available: http://www.drdobbs.com/architecture-and-design/why-we-need-a-theory-for-software-engine/220300840

[4] J. Sweller, "Cognitive Load During Problem Solving: Effects on Learning," *Cognitive Science*, vol. 12, no. 2, pp. 257–285, Apr. 1988. [Online]. Available: http://dx.doi.org/10.1207/s15516709cog1202_4

[5] J. Sweller and P. Chandler, "Evidence for cognitive load theory," *Cognition and instruction*, vol. 8, no. 4, pp. 351–362, 1991.

[6] G. A. MILLER, "The magical number seven plus or minus two: some limits on our capacity for processing information." *Psychological review*, vol. 63, no. 2, pp. 81–97, Mar. 1956. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/13310704

[7] A. Baddeley, "Working memory: looking back and looking forward," *Nature Reviews. Neuroscience*, vol. 4, no. 10, p. 829, 2003. [Online]. Available: https://labs.wsu.edu/attention-perception-performance/documents/2016/05/baddeley_review_2003.pdf

[8] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976. [Online]. Available: http://juacompe.mrchoke.com/natty/thesis/FrameworkComparison/A%20complexity%20measure.pdf

[9] M. H. Halstead, *Elements of software science*, ser. Elsevier computer science library : operational programming systems series. New York, NY: North-Holland, 1977.

[10] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: a software science validation," *Software Engineering, IEEE Transactions on*, no. 6, pp. 639–648, 1983.

[11] Y. Wang, "On the cognitive complexity of software and its quantification and formal measurement," *International Journal of Software Science and Computational Intelligence (IJSSCI)*, vol. 1, no. 2, pp. 31–53, 2009.

[12] J. Shao and Y. Wang, "A new measure of software complexity based on cognitive weights," *Electrical and Computer Engineering, Canadian Journal of*, vol. 28, no. 2, pp. 69–74, 2003.

[13] P. Naur and B. Randell, Eds., *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969. [Online]. Available: http://portal.acm.org/citation.cfm?id=1102020

[14] B. Moseley and P. Marks, "Out of the Tar Pit," *Complexity*. [Online]. Available: http://shaffner.us/cs/papers/tarpit.pdf

[15] S. Hastie and S. Wojewoda. (2015, Oct.) Standish Group 2015 Chaos Report - Q&A with Jennifer Lynch. [Online]. Available: https://www.infoq.com/articles/standish-chaos-2015

[16] M. Jørgensen and K. Moløkken-Østvold, "How large are software cost overruns? A review of the 1994 CHAOS report," *Information and Software Technology*, vol. 48, no. 4, pp. 297–301, Apr. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2005.07.002

[17] R. L. Glass, "The Standish report: does it really describe a software crisis?" *Communications of the ACM*, vol. 49, no. 8, pp. 15–16, Aug. 2006. [Online]. Available: http://dx.doi.org/10.1145/1145287.1145301

[18] R. C. Martin, J. O. Coplien, K. Wampler, J. W. Grenning, B. L. Schuchert, J. Langr, T. R. Ottinger, and M. C. Feathers, *Clean code : a handbook of agile software craftsmanship*, 1st ed. Prentice Hall, Aug. 2016. [Online]. Available: http://www.worldcat.org/isbn/0132350882

[19] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mallor, K. Shwaber, and J. Sutherland, "The Agile Manifesto," The Agile Alliance, Tech. Rep., 2001.

[20] K. Beck, *Test Driven Development: By Example*, 1st ed. Addison-Wesley Professional, Nov. 2002. [Online]. Available: http://www.worldcat.org/isbn/0321146530

[21] E. Hendrickson, "Driving Development with Tests: ATDD and TDD," in *STAR-WEST*. Software Quality Engineering, Oct. 2008.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov. 1994. [Online]. Available: http://www.worldcat.org/isbn/0201633612

[23] E. Freeman, B. Bates, K. Sierra, and E. Robson, *Head First Design Patterns: A Brain-Friendly Guide*, 1st ed. O'Reilly Media, Nov. 2004. [Online]. Available: http://www.worldcat.org/isbn/0596007124

[24] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston, *Object-Oriented Analysis and Design with Applications (3rd Edition)*, 3rd ed. Addison-Wesley Professional, Apr. 2007. [Online]. Available: http://www.worldcat.org/isbn/020189551X

[25] A. Shalloway and J. R. Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design (2nd Edition)*, 2nd ed. Addison-Wesley Professional, Oct. 2004. [Online]. Available: http://www.worldcat.org/isbn/0321247140

[26] K. Beck, W. Cunningham, and W. S. Services. (1989) A laboratory for teaching object-oriented thinking. http://c2.com/doc/oopsla89/paper.html. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.4074

[27] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, 1st ed. Prentice Hall, Jun. 1990. [Online]. Available: http://www.worldcat.org/isbn/0136298257

[28] G. Wilson, "What We Actually Know About Software Development, and Why We Believe It's True," in *CUSEC*, 2010.

[29] R. A. Tarmizi and J. Sweller, "Guidance during mathematical problem solving." *Journal of educational psychology*, vol. 80, no. 4, p. 424, 1988.

[30] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)*, anniversary ed. Addison-Wesley Professional, Aug. 1995. [Online]. Available: http://www.worldcat.org/isbn/0201835959

[31] K. Beck, *Extreme Programming Explained: Embrace Change*, us ed ed. Addison-Wesley Professional, Oct. 1999. [Online]. Available: http://www.worldcat.org/isbn/0201616416

[32] P. McBreen, *Software Craftsmanship: The New Imperative*, 1st ed. Addison-Wesley Professional, Sep. 2001. [Online]. Available: http://www.worldcat.org/isbn/0201733862

[33] E. W. Dijkstra, "Letters to the Editor: Go to Statement Considered Harmful," *Commun. ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968. [Online]. Available: http://dx.doi.org/10.1145/362929.362947

[34] ——, "Notes on structured programming. 1969, appeared in OJ Dahl, EW Dijkstra, and CAR Hoare (eds): Structured Programming," 1972.

[35] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall, Apr. 1988. [Online]. Available: http://www.worldcat.org/isbn/0131103628

[36] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972. [Online]. Available: http://dx.doi.org/10.1145/361598.361623

[37] A. C. Kay, "The Early History of Smalltalk," *SIGPLAN Not.*, vol. 28, no. 3, pp. 69–95, Mar. 1993. [Online]. Available: http://doi.acm.org/10.1145/155360.155364

[38] D. E. Knuth, *Literate Programming (Center for the Study of Language and Information - Lecture Notes)*, 1st ed. Center for the Study of Language and Inf, Jun. 1992. [Online]. Available: http://www.worldcat.org/isbn/0937073806

[39] ——, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, Jan. 1984. [Online]. Available: http://comjnl.oxfordjournals.org/content/27/2/97.full.pdf+html

[40] N. Ramsey, "Literate Programming Simplified," *IEEE Softw.*, vol. 11, no. 5, pp. 97–105, Sep. 1994. [Online]. Available: http://dx.doi.org/10.1109/52.311070

[41] G. White and M. Sivitanides, "Cognitive Differences Between Procedural Programming and Object Oriented Programming," *Information Technology and Management*, vol. 6, no. 4, pp. 333–350, Oct. 2005.

[42] C. Douce, "The stores model of code cognition," in *In Programmer Psychology Interest Group*, 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.222.9075

[43] B. Curtis, "Measurement and experimentation in software engineering," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1144–1157, 1980.

[44] B. Curtis, I. Forman, R. Brooks, E. Soloway, and K. Ehrlich, "Psychological perspectives for software science," *Information Processing & Management*, vol. 20, no. 1-2, pp. 81–96, Jan. 1984. [Online]. Available: http://dx.doi.org/10.1016/0306-4573(84)90041-4

[45] R. C. Clark, F. Nguyen, and J. Sweller, *Efficiency in Learning: Evidence-Based Guidelines to Manage Cognitive Load*. Pfeiffer, Dec. 2005. [Online]. Available: http://www.worldcat.org/isbn/0787977284

[46] N. Hollender, C. Hofmann, M. Deneke, and B. Schmitz, "Integrating cognitive load theory and concepts of human–computer interaction," *Computers in Human Behavior*, vol. 26, no. 6, pp. 1278–1288, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.chb.2010.05.031

[47] S. Oviatt, "Human-centered Design Meets Cognitive Load Theory: Designing Interfaces That Help People Think," in *Proceedings of the 14th Annual ACM International Conference on Multimedia*, ser. MULTIMEDIA '06. New York, NY, USA: ACM, 2006, pp. 871–880. [Online]. Available: http://dx.doi.org/10.1145/1180639.1180831

[48] J. Rumbaugh, R. Jacobson, and G. Booch, *The Unified Modelling Language Reference Manual*, 1st ed. Addison-Wesley, Jan. 1999. [Online]. Available: www.amazon.co.uk/exec/obidos/ASIN/020130998X/026-2174472-9898019

[49] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, 1st ed. Addison-Wesley Professional, Feb. 1999. [Online]. Available: http://www.worldcat.org/isbn/0201571692

[50] K. Beck, *Implementation Patterns*, 1st ed. Addison-Wesley Professional, Nov. 2007. [Online]. Available: http://www.worldcat.org/isbn/0321413091

[51] M. Fowler, *Patterns of Enterprise Application Architecture*, 1st ed. Addison-Wesley Professional, Nov. 2002. [Online]. Available: http://www.worldcat.org/isbn/0321127420

[52] T. Parr, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers)*, 1st ed. Pragmatic Bookshelf, Jan. 2010. [Online]. Available: http://www.worldcat.org/isbn/193435645X

[53] L. Rising and N. S. Janoff, "The Scrum software development process for small teams," *Software, IEEE*, vol. 17, no. 4, pp. 26–32, Jul. 2000. [Online]. Available: http://dx.doi.org/10.1109/52.854065

[54] D. J. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, Apr. 2010. [Online]. Available: http://www.worldcat.org/isbn/0984521402

[55] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, 1st ed. Addison-Wesley Professional, Oct. 1999. [Online]. Available: http://www.worldcat.org/isbn/020161622X

[56] M. Fowler and K. Beck, *Refactoring improving the design of existing code*, 1st ed. Addison-Wesley, Jul. 2013. [Online]. Available: http://www.worldcat.org/isbn/0201485672

[57] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992. [Online]. Available: http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7302/ReadingMaterial/Opdyke92.pdf

[58] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, 1st ed. Prentice Hall, Oct. 2002. [Online]. Available: http://www.worldcat.org/isbn/0135974445

[59] B. Liskov, "Keynote Address - Data Abstraction and Hierarchy," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, ser. OOPSLA '87. New York, NY, USA: ACM, 1987, pp. 17–34. [Online]. Available: http://dx.doi.org/10.1145/62138.62141

[60] B. Meyer, *Object-Oriented Software Construction (Book/CD-ROM) (2nd Edition)*, 2nd ed. Prentice Hall, Apr. 1997. [Online]. Available: http://www.worldcat.org/isbn/0136291554

[61] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, Eds., *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Piscataway, NJ, USA: IEEE Press, 2001.

[62] F. DeRemer and H. Kron, "Programming-in-the Large Versus Programming-in-the-small," *SIGPLAN Not.*, vol. 10, no. 6, pp. 114–121, Apr. 1975. [Online]. Available: http://dx.doi.org/10.1145/390016.808431

[63] A. Hunt, *Pragmatic Thinking and Learning: Refactor Your Wetware (Pragmatic Programmers)*, 1st ed. Pragmatic Bookshelf, Nov. 2008. [Online]. Available: http://www.worldcat.org/isbn/1934356050

[64] C. Alexander, *The timeless way of building*. New York: Oxford University Press, 1979, vol. 1.

[65] P. Chandler and J. Sweller, "The split-attention effect as a factor in the design of instruction," *British Journal of Educational Psychology*, vol. 62, no. 2, pp. 233–246, 1992.

[66] S. Kalyuga, P. Ayres, P. Chandler, and J. Sweller, "The Expertise Reversal Effect," *Educational Psychologist*, vol. 38, no. 1, pp. 23–31, Mar. 2003. [Online]. Available: http://dx.doi.org/10.1207/s15326985ep3801_4

[67] K. Pugh, *Interface-Oriented Design (Pragmatic Programmers)*. Pragmatic Bookshelf, 2006.

[68] F. Paas, J. E. Tuovinen, H. Tabbers, and P. W. M. Van Gerven, "Cognitive load measurement as a means to advance cognitive load theory," *Educational psychologist*, vol. 38, no. 1, pp. 63–71, 2003.

[69] F. G. W. C. Paas, J. J. G. Van Merriënboer, and J. J. Adam, "Measurement of cognitive load in instructional research," *Perceptual and motor skills*, vol. 79, no. 1, pp. 419–430, 1994. [Online]. Available: https://www.researchgate.net/profile/Fred_Paas/publication/15390085_Measurement_of_cognitive_load_in_instructional_research/links/55ad2fdb08ae98e661a41759/Measurement-of-cognitive-load-in-instructional-research.pdf

[70] R. Brunken, J. L. Plass, and D. Leutner, "Direct measurement of cognitive load in multimedia learning," *Educational psychologist*, vol. 38, no. 1, pp. 53–61, 2003. [Online]. Available: http://steinhardtapps.es.its.nyu.edu/create/courses/2174/reading/Bruenken_Plass_Leutner_EP.pdf

[71] A. Degroot, *Thought and Choice in Chess (Psychological Studies)*, 2nd ed. Mouton De Gruyter, 1968. [Online]. Available: http://www.worldcat.org/isbn/9027979146

[72] D. Egan and B. Schwartz, "Chunking in recall of symbolic drawings," vol. 7, no. 2, pp. 149–158, 1979. [Online]. Available: http://dx.doi.org/10.3758/bf03197595

[73] P. Benner, "Using the Dreyfus Model of Skill Acquisition to Describe and Interpret Skill Acquisition and Clinical Judgment in Nursing Practice and Education," *Bulletin of Science, Technology & Society*, vol. 24, no. 3, pp. 188–199, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1177/0270467604265061

[74] K. Ehrlich and E. Soloway, "Human Factors in Computer Systems," J. C. Thomas and M. L. Schneider, Eds. Norwood, NJ, USA: Ablex Publishing Corp., 1984, ch. An Empirical Investigation of the Tacit Plan Knowledge in Programming, pp. 113–133. [Online]. Available: http://portal.acm.org/citation.cfm?id=823

[75] R. S. Rist, "Plans in Programming: Definition, Demonstration, and Development," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Norwood, NJ, USA: Ablex Publishing Corp., 1986, pp. 28–47. [Online]. Available: http://portal.acm.org/citation.cfm?id=28884

[76] S. E. Dreyfus and H. L. Dreyfus, "A five-stage model of the mental activities involved in directed skill acquisition," DTIC Document, Tech. Rep., 1980.

[77] Y.-H. Choe, C.-Y. Jong, and S. Han, "Software cognitive information measure based on relation between structures," *arXiv preprint arXiv:1304.0374*, 2013. [Online]. Available: https://arxiv.org/pdf/1304.0374.pdf

[78] L. Carlson. (2014) Career Path of a Programmer. https://www.ctl.io/developers/blog/post/career-path-of-a-programmer/. [Online]. Available: https://www.ctl.io/developers/blog/post/career-path-of-a-programmer/

[79] J. Atwood. (2008) The Years of Experience Myth. https://blog.codinghorror.com/the-years-of-experience-myth/. [Online]. Available: https://blog.codinghorror.com/the-years-of-experience-myth/

[80] T. Mastaglio and J. Rieman, "How experts infer novice programmer expertise: a protocol analysis of LISP code evaluation," in *Proc. Empirical Studies of Programmers, Fourth Workshop*, 1991, pp. 177–195.

[81] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on software engineering*, no. 5, pp. 595–609, 1984.

[82] P. Graham, "Beating the Averages," 2001. [Online]. Available: http://www.paulgraham.com/avg.html

[83] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

APPENDIX A

LIKERT SCALE FOR CONTROL STUDY

Cognitive Code Study Code Paths Likert Scale A Likert Scale is a psychometric scale that is commonly used in Cognitive Load Theory driven trials to gauge the subjective level of effort someone feels they need to apply to understand a given stimulus.

1. Email address *

---

2. How hard is this method to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

3. How hard is this method to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

4. How hard is this method to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

5. How hard is this method to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

6. How hard is this method to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

7. How hard is this method to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

   ---

8. How hard is this method to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

   ---

9. How hard is this method to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

   ---

10. How hard is this method to understand? Mark only one oval.

    1 2 3 4 5 6 7

    Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

    ---

11. How hard is this method to understand? Mark only one oval.

    1 2 3 4 5 6 7

    Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

☐ Send me a copy of my responses.

APPENDIX B

LIKERT SCALE FOR EXPERIMENTAL STUDY

Cognitive Code Study Code Paths Likert Scale A Likert Scale is a psychometric scale that is commonly used in Cognitive Load Theory driven trials to gauge the subjective level of effort someone feels they need to apply to understand a given stimulus.

1. Email address *

---

2. How hard is this code to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

3. How hard is this code to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

4. How hard is this code to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

5. How hard is this code to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

6. How hard is this code to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

7. How hard is this code to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

8. How hard is this code to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

9. How hard is this code to understand? Mark only one oval.

   1 2 3 4 5 6 7

   Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

10. How hard is this code to understand? Mark only one oval.

    1 2 3 4 5 6 7

    Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

11. How hard is this code to understand? Mark only one oval.

    1 2 3 4 5 6 7

    Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

12. How hard is this code to understand? Mark only one oval.

    1 2 3 4 5 6 7

    Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

13. How hard is this code to understand? Mark only one oval.

    1 2 3 4 5 6 7

    Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

14. How hard is this code to understand? Mark only one oval.

1 2 3 4 5 6 7

Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

15. How hard is this code to understand? Mark only one oval.

1 2 3 4 5 6 7

Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

16. How hard is this code to understand? Mark only one oval.

1 2 3 4 5 6 7

Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

---

17. How hard is this code to understand? Mark only one oval.

1 2 3 4 5 6 7

Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

18. How hard is this code to understand? Mark only one oval.

1 2 3 4 5 6 7

Very easy ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ ⬭ Very hard

APPENDIX C

MEASURING THE COGNITIVE LOAD OF CODE:

Investigating the use of CLT techniques to minimize debugging time and errors

C.1 Review

As I established at the beginning of this dissertation, Cognitive Load Theory is a fruitful field of principles from instructional and user interface design that lies fallow for software engineering. Professional software engineering involves many non-programming related activities, but the core unifying glue is always code. As software engineering practitioner literature has argued, code that is poorly written and difficult to understand can cost companies time, money, and even–such as in the case of the HEARTBLEED OpenSSL bug– expose massive security vulnerabilities just due to a missing set of " " for an if-statement.

Code style is arguably important, but attempts to analytically quantify the impact of different techniques remains more art than science. Additionally, researchers such as Lionel Briad and Ivar Jacobson have decried the lack of strong theoretical principles for software engineering. This work explores the intersection of Cognitive Load Theory and software engineering best practices to answer the question: "Is there a relationship between evidence driven best practices for organizing educational content and published best practices for writing code?" Experiment: Fix a bug in date-time manipulation in Joda Time I chose to apply software engineering best practice techniques identified in Clean Code and Refactoring to a popular open-source Java library called Joda-time. Joda-time is one of the most highly-used libraries in the Java ecosystem. Jodatime sits in the dependency tree of many popular software packages such as Spring, the Lift web application framework, Hibernate annotations, and many, many more. As such, this is not a contrived exercise dreamed up by a graduate student in a lab to show an effect. This is real, battle-tested, used by millions code.

I also chose Joda-time because it is written in Java. Java and C++ are the most well-known programming languages in the community. Java is taught in many college curriculums; finding familiar programmers is less difficult than other languages. Joda-time solves a very general problem–date & time manipulation. Most languages have date libraries for common operations like arithmetic, formatting, or timezone conversion. Good and usable libraries provide broad, usable abstractions for complicated functionality. Date & Time manipulation is very complicated, often causing bugs even in major professional software engineering environments.

C.2 Bug: ISO8601 Years

The "devil in the detail" is all of the edge cases. One of the devilish details is the existence of a large, amorphous, ever-changing set of formats describing a date. The International Standards Organization attempted to ameliorate this with ISO8601, which defines a canonical format for the interchange of dates and times across locales. One of the quirks of the standard is that the published format is YYYY-MM-DD,

however, the standard allows for more than 4-digit years in cases where sender and receiver have agreed upon extra digits by prepending with a + or - [± YYYYY].

Jodatime did not originally account for this. This led to https://github.com/JodaOrg/joda-time/issues/86. Hari Shankar explored the depths of the code and identified an issue and a fix. He did not report how long it took to find the bug, or how difficult it was to come up with a solution. When I first found this issue, it was still open. In October of 2015, Steven Colebourne (primary author/maintainer of the library) merged in Shankar's fix.

C.3 Analysis of Accepted Solution

The accepted solution follows a common maintenance programmer practice: change as little code as possible and duplicate where necessary and with slight tweaks to achieve desired behavior. It is not exactly but similar in vein to "Programming By Difference." It is a conservative approach that tends to be favored when code is in obsolescence, has a large number of consumers, or is otherwise hard to change.

"Other reasons" that code can be hard to change include a lack of confidence in correct behavior, or difficulty in understanding algorithms and data structures written by someone else. The commit that introduced the fix included unit test cases that manifest the bug (though notably only try 5 digit years, perhaps not fully exercising failure modes), suggesting the maintainer understood the code enough to augment the existing test suite, a software best practice. The original code was not written solely by Stephen Colebourne. Attribution information in the comments suggests it was a collaborative effort between Colebourne, Brian S. O'Neill, and Fredrik Borgh. It's possible this had an effect on the aggressiveness of the fix.

Using SonarQube 5.4 for analysis, we find that the DateTimeFormatterBuilder alone has 1600 LoC. 2625 total lines (including comments) with 162 issues found via static source analysis, with 2.3% duplications, estimated to take 2.5 days to fix. It has a Cyclomatic Complexity score of 550, the highest in the library.

When gathering statistics of the library with the "Run Tests with Coverage" option of IDEA IntelliJ, Joda Time exhibits an impressing 98% class/91% line/90% method coverage. Such a high level of coverage potentially provides developers the option to experiment with more drastic structural Refactoring and verify existing behavior, but time and other issues may not always make such re-architecture possible.

C.4 Experiment: contrast debugging time and performance of accepted solution versus CLT optimized

For the purposes of this experiment, we hold the state of JodaTime as reflected in Git commit on August 2nd 2015 as the "control" group, not including the given fix. We use the provided fix as the "gold standard", as it was accepted by the core library contributor. We follow a 2x2 factorial design where we account for novice/expert and control/experimental for debugging time, number of defects introduced as measured by failing tests, and perceived cognitive load as measured on our Likert Scale.

With such setup in mind, I will proceed to show how I applied software engineering best practices influenced by Cognitive Load Theory to develop the experimental treatment.

C.5 Development of Experimental Version

C.5.1 Starting small: sequencing, chunking, and intrinsic complexity at the variable and method level

I began with trying to simplify a complicated if-statement that included assignment, array indexing, and bounding checking in one compound predicate. I was guided by the presence of a comment. Robert Martin's Clean Code suggests that "Comments are Lies." They often become out of date as code changes with time.

Comments also signal a failure to communicate, a missed opportunity for the code to be self-descriptive. A programmer who reads a comment incurs the additional cognitive load of reading the comment, having to verify whether it accurately summarizes the situation, and maintaining a mental mapping between the descriptive comment and the cryptic code. Cognitive Load Theory shows that redundant information adds extraneous cognitive load–learning outcomes for material that has a picture and paragraph repetitively describing the same phenomena are generally improved if the information is concisely expressed in one form. Thus, we use the technique of INTRODUCE EXPLAINING VARIABLE to make the conditional more descriptive. Realizing that these temporary variables had added lines to answer a limited scope question within the body of a method that was already 64 lines long, increasing Germane Cognitive Load for understanding the full execution of this method, I used the CLT technique of managing Germane Cognitive Load by sequencing and chunking, applying REPLACE TEMP WITH QUERY to push knowledge of this question into a cohesive chunk, a method.

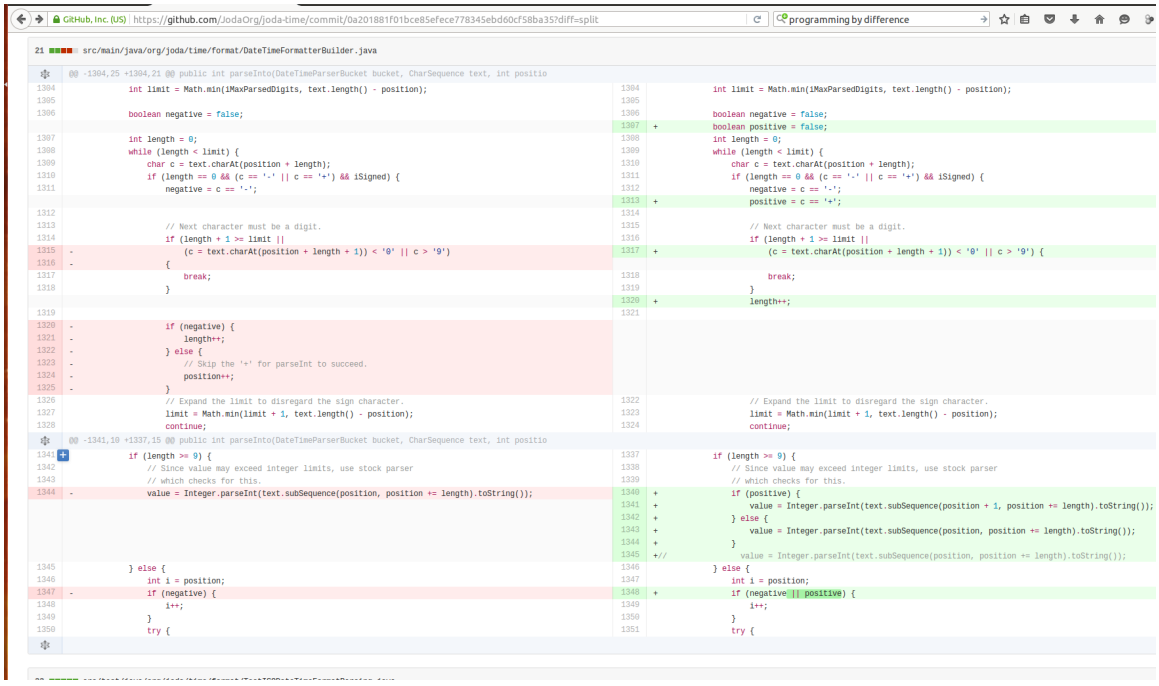


Figure C.1: insert caption

When I ran the tests, I discovered something troubling: a test was broken. The behavior had changed.

```
Tests run: 4161, Failures: 3, Errors: 1, Skipped: 0, Time elapsed:
6.298 sec <<<FAILURE! - in org.joda.time.TestAllPackages
testFormat_year(org.joda.time.format.TestDateTimeFormat) Time elapsed:
0.015 sec <<<ERROR!
java.lang.StringIndexOutOfBoundsException: String index out of range:
1
    at java.lang.String.charAt(String.java:646)
    at org.joda.time.format.DateTimeFormatterBuilder$NumberFormatter.parseInto(Date
    at org.joda.time.format.DateTimeFormatter.parseDateTime(DateTimeFormatter.java:
    at org.joda.time.format.TestDateTimeFormat.testFormat_year(TestDateTimeFormat.j
```

I stared at the code for a good 20 minutes, not understanding what went wrong. Opdyke defines Refactoring as "structural code changes that exhibit no external behavioral difference." Clearly what I'd done had ended up not being Refactoring, despite the fact that I'd leveraged the baked in functionality of IDEA IntelliJ Ultimate Edition 14, which includes a robust static analysis checker that usually detects such changes. Pouring over the expression hardly helped, the change "looked safe." Attaching a debugger and stepping through the broken test revealed the problem.

The predicate of the if statement had caught my eye for having a lot of complexity and a comment to describe it. A hidden complexity that wasn't immediately obvious to me was that it relied on the short-circuit evaluation behavior of the —— operator in Java as an implicit GUARD CLAUSE

```
length + 1 >= limit || (c = text.charAt(position + length + 1)) <'0'
|| c >'9')
```

In the previous expression, the `length + 1 >= limit` prevents the `text.charAt(position + length + 1)` from being evaluated when `length + 1` would result trying to index a character outside of the string, causing an error. Thus, there is an additional semantic that provides safety based on the ordering of expressions.

This is exactly the type of hidden complexity that creates anxiety in maintenance engineers in modifying existing code. It is typified by reliance on a subtle semantic provided by the language runtime. This can be considered germane cognitive load– language semantics are part of a programmer's toolkit. Expert programmers often use these kinds of techniques, as experts are shown to better handle an increase in germane cognitive load. Novices, however, may experience information overload.

I can simplify this by rewriting the code to be a little longer, but more explicit.

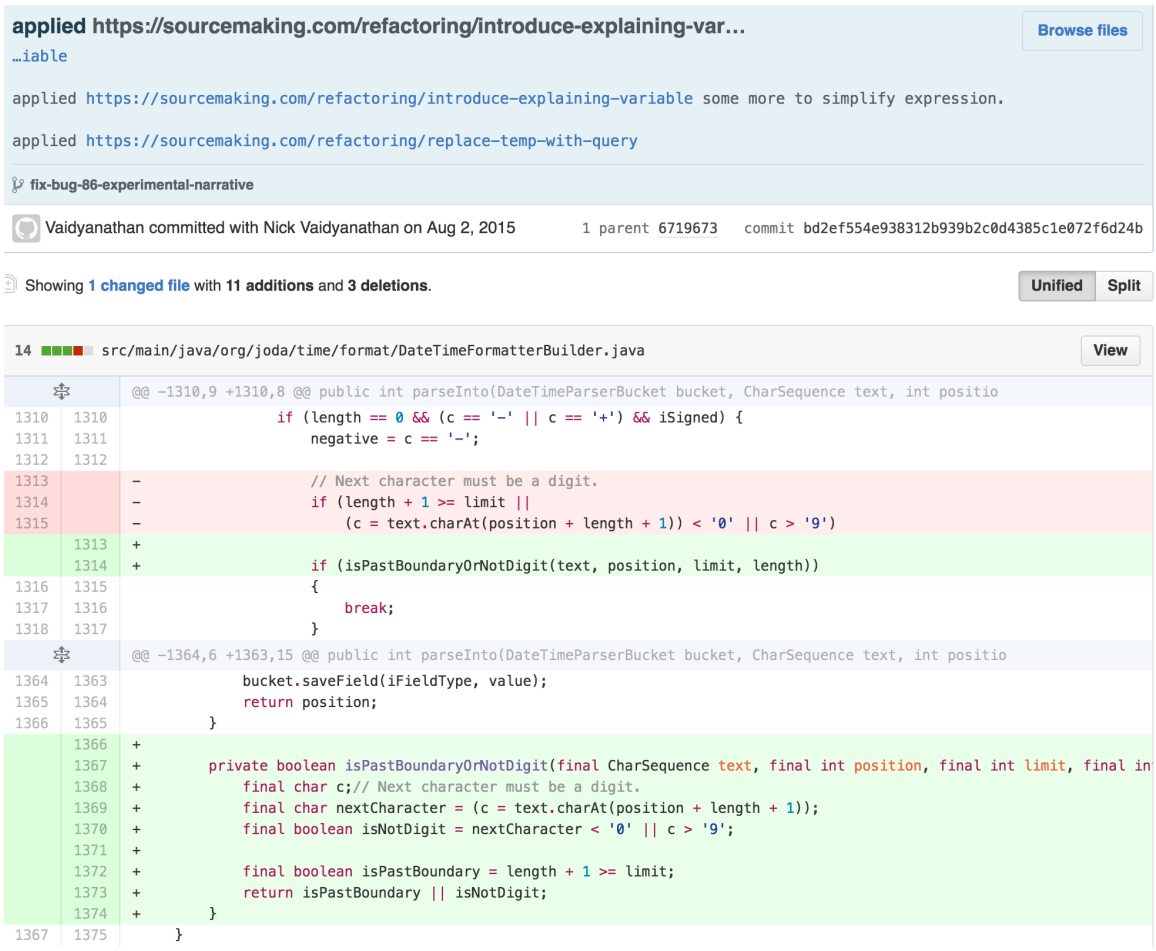


Figure C.2: insert caption

In this example, I make the GUARD CLAUSE more explicit and match the standard form, and use an explanatory named variables to describe the why of the calculation's what.

C.5.2 Building up: moving on to the class level

Returning my attention to the original expression, I apply the same INTRODUCE EXPLANATORY VARIABLE and EXTRACT METHOD technique.:

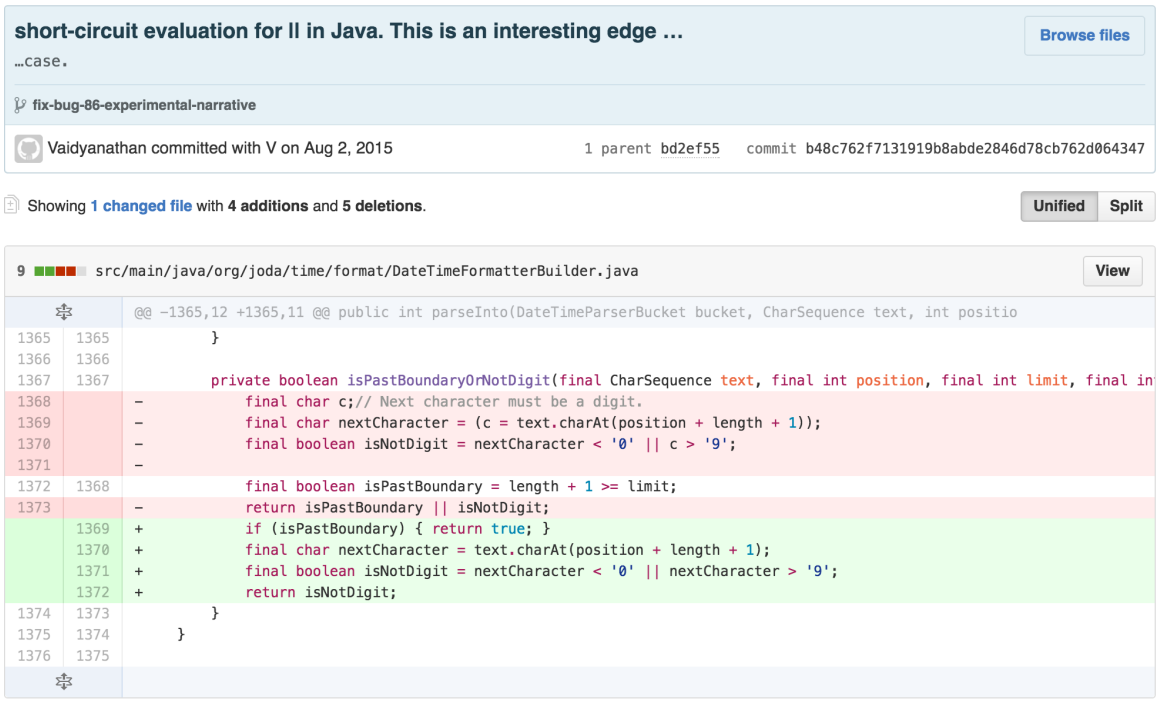


Figure C.3: insert caption

I also made the chunked methods static to signify that they were "pure functions." They had no reliance on the `DateTimeFormatterBuilder`'s internal state and solely exist as calculations. This structural change helped me realize that much of the body of the `parse` method was isolated from the rest of the class. This is often an opportune situation to apply REPLACE METHOD WITH METHOD OBJECT, which can decrease the length and intrinsic cognitive load of the original class via chunking its content with the created collaborating class.

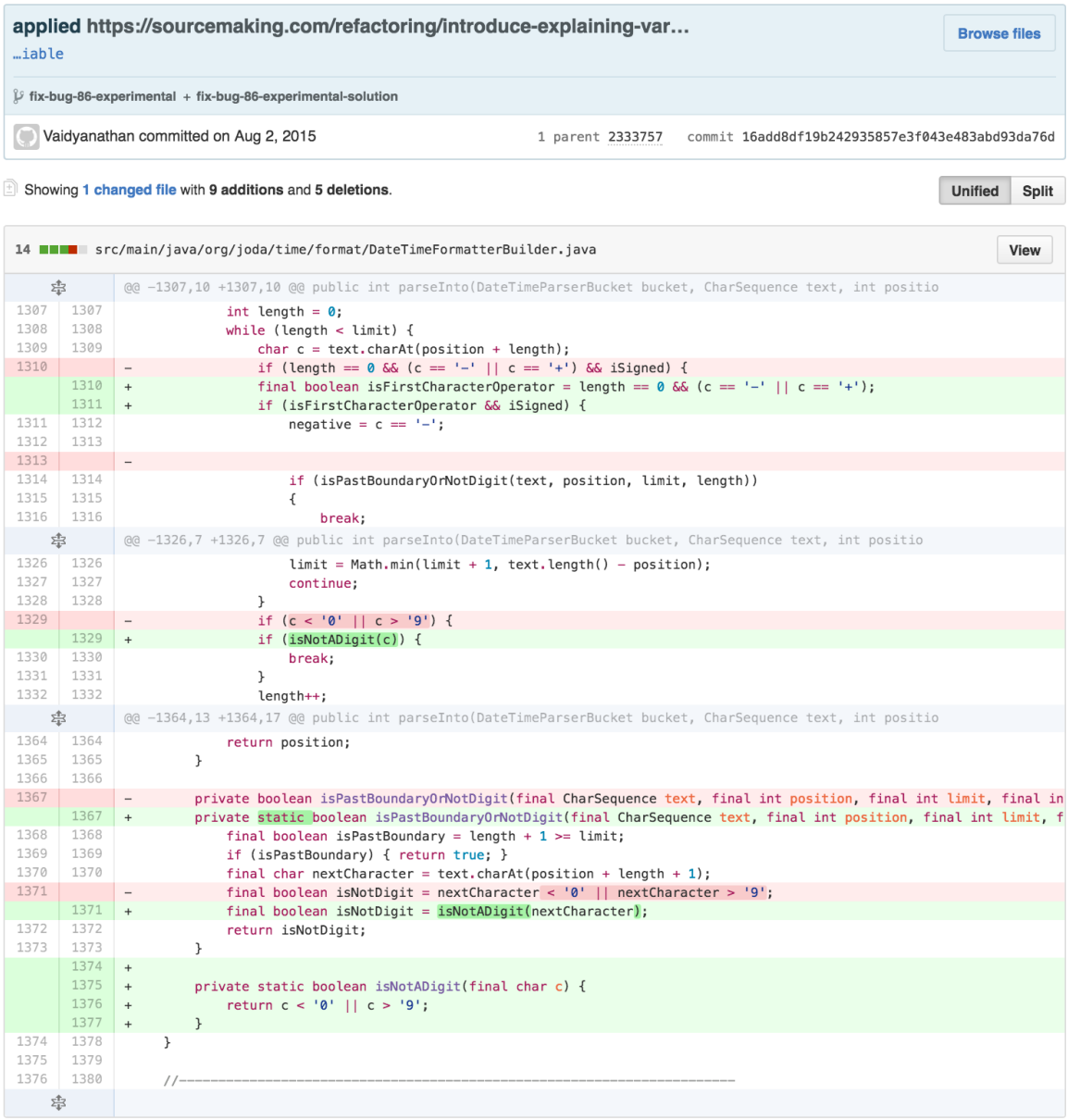


Figure C.4: insert caption

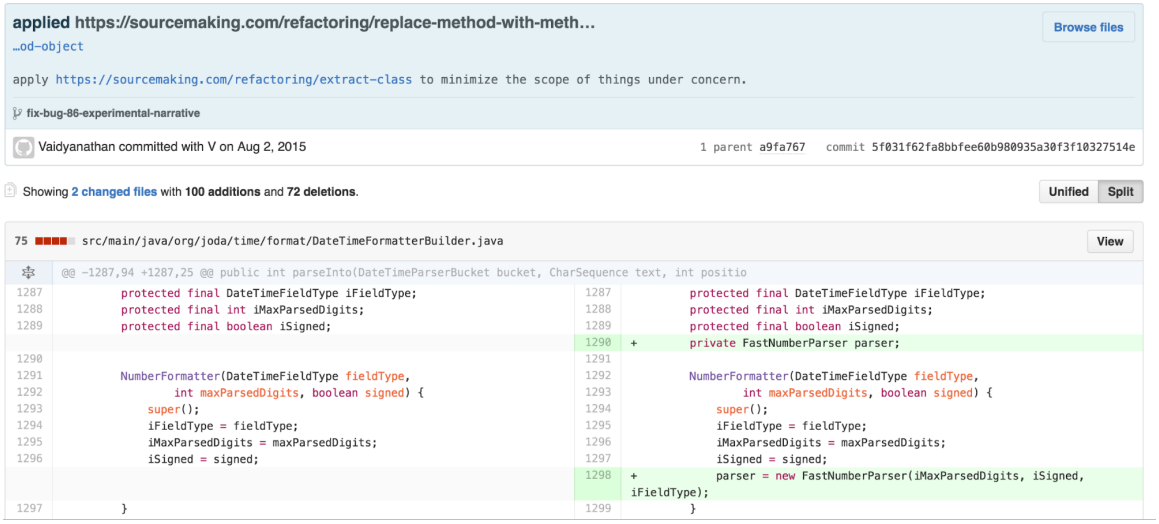


Figure C.5: insert caption

Figure C.6: insert caption

Figure C.7: insert caption

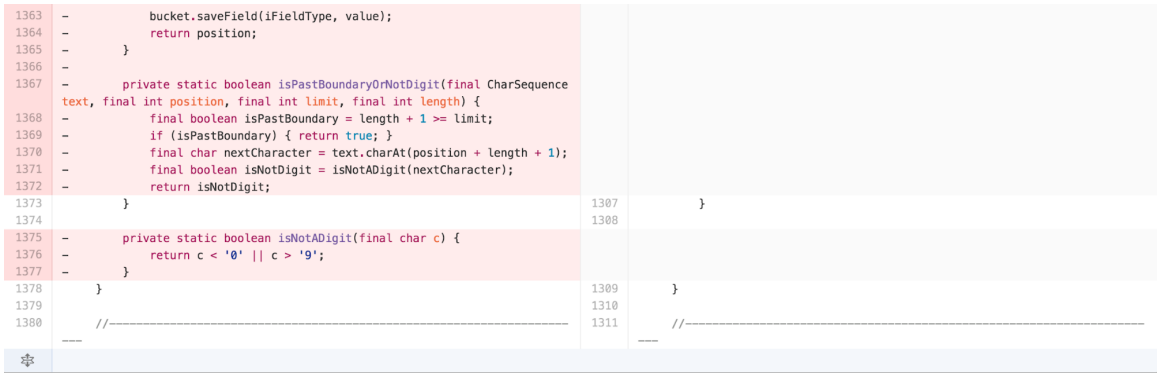


Figure C.8: insert caption

97 ■■■■■ src/main/java/org/joda/time/format/FastNumberParser.java View

```
@@ -0,0 +1,97 @@
+package org.joda.time.format;
+
+import org.joda.time.DateTimeFieldType;
+import org.joda.time.format.DateTimeParserBucket;
+
+final class FastNumberParser {
+    private final DateTimeFieldType iFieldType;
+    private final int iMaxParsedDigits;
+    private final boolean iSigned;
+
+    public FastNumberParser(final int maxParsedDigits,
+                            final boolean signed,
+                            final DateTimeFieldType fieldType) {
+        iFieldType = fieldType;
+        iMaxParsedDigits = maxParsedDigits;
+        iSigned = signed;
+    }
+
+    public int parse(final DateTimeParserBucket bucket,
+                     final CharSequence text,
+                     int position) {
+        int limit = Math.min(iMaxParsedDigits, text.length() - position);
+
+        boolean negative = false;
+        int length = 0;
+        while (length < limit) {
+            char c = text.charAt(position + length);
+            final boolean isFirstCharacterOperator = length == 0 && (c == '-' || c == '+');
+            if (isFirstCharacterOperator && iSigned) {
+                negative = c == '-';
+
+                if (isPastBoundaryOrNotDigit(text, position, limit, length)) {
+                    break;
+                }
+
+
```

Figure C.9: insert caption

```
+                if (negative) {
+                    length++;
+                } else {
+                    // Skip the '+' for parseInt to succeed.
+                    position++;
+                }
+                // Expand the limit to disregard the sign character.
+                limit = Math.min(limit + 1, text.length() - position);
+                continue;
+            }
+            if (isNotADigit(c)) {
+                break;
+            }
+            length++;
+        }
+
+        if (length == 0) {
+            return ~position;
+        }
+
+        int value;
+        if (length >= 9) {
+            // Since value may exceed integer limits, use stock parser
+            // which checks for this.
+            value = Integer.parseInt(text.subSequence(position, position += length).toString());
+        } else {
+            int i = position;
+            if (negative) {
+                i++;
+            }
+            try {
+                value = text.charAt(i++) - '0';
+            } catch (StringIndexOutOfBoundsException e) {
+                return ~position;
+            }
+            position += length;
+            while (i < position) {
+                value = ((value << 3) + (value << 1)) + text.charAt(i++) - '0';
+            }
+            if (negative) {
+                value = -value;
```

Figure C.10: insert caption

```
+            }
+        }
+
+        bucket.saveField(iFieldType, value);
+        return position;
+    }
+
+    private static boolean isPastBoundaryOrNotDigit(final CharSequence text, final int position, final int limit, final int lengt
+        final boolean isPastBoundary = length + 1 >= limit;
+        if (isPastBoundary) {
+            return true;
+        }
+        final char nextCharacter = text.charAt(position + length + 1);
+        final boolean isNotDigit = isNotADigit(nextCharacter);
+        return isNotDigit;
+    }
+
+    private static boolean isNotADigit(final char c) {
+        return c < '0' || c > '9';
+    }
+}
```

Figure C.11: insert caption

The new class is 97 lines long, with the majority of the work happening in the 63

line long `parse` method. The result of `parse` can be tested and understood independently of details from the `NumberFormatter` inside of `DateTimeFormatterBuilder`.

C.6 Reducing Control Flow Complexity

Some might argue that a 63 line method is "simple enough." Working heuristics of working memory, however, suggest that the human brain can fit between 4-9 "chunks" in memory at a particular time. This aligns well with the guidance in Clean Code that "Functions should do One Thing" and "Functions should be small". Consequently, it's worth turning attention to simplifying the `parse` method.

The cognitive complexity of the function is compounded by a variety of control structures. There are 2 while loops, 9 if statements, and 3 return statements. Extracting methods out of such a function is difficult because the multiple control flow breaks and updates to local variables require reasoning that is beyond many static analysis code tools, such as the Refactoring tools in IDEA Intellij. Often it helps to start "from the bottom up", simplifying small blocks of complicated control structures until the tangled web of control flow clarifies.

First, a little cleanup. I can remove an unused import, as imports add to the cognitive load of a class by signaling interconnection with other components of a system.

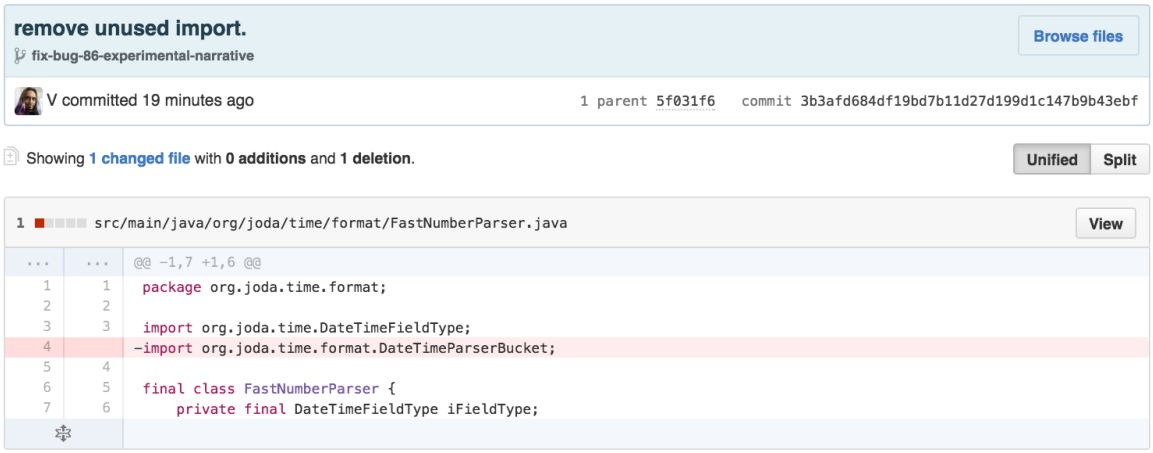


Figure C.12: insert caption

Next, I simplify according to the Principle of Least Astonishment by replacing the use of a StringIndexOutOfBoundsException for flow control (commonly argued in the literature) with a GUARD CLAUSE.

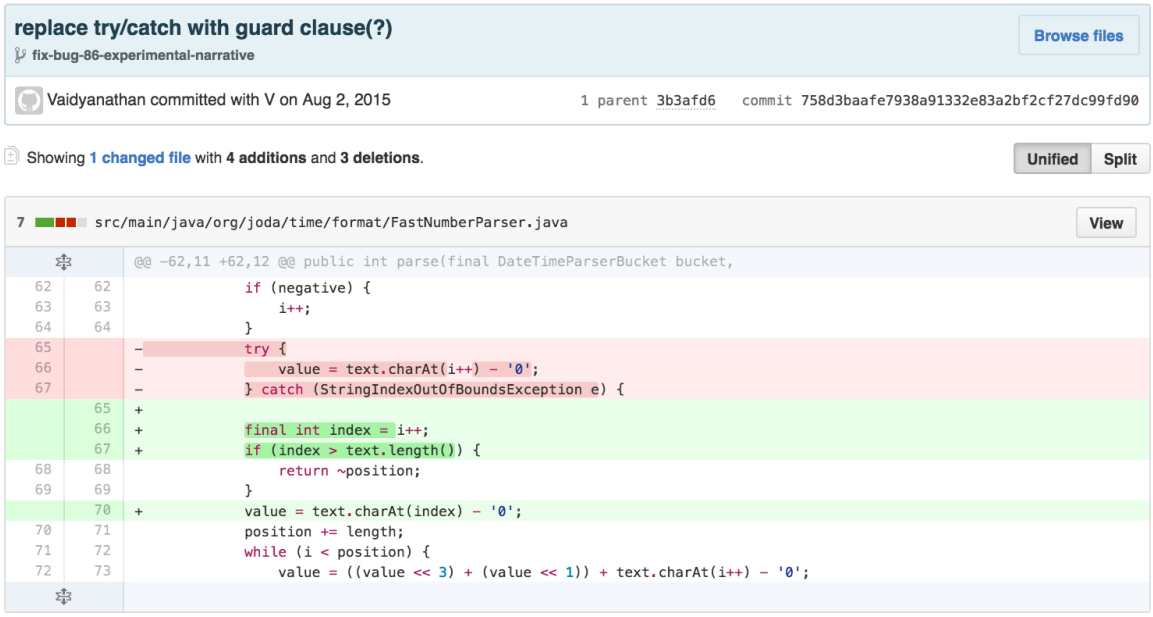


Figure C.13: insert caption

The body of `parse` has a strong visual indicator of separation between the first while loop, which updates the `position` and `length` variables, and a second "section" that calculates the value. In order to be able to break out methods effectively, I want to simplify the control flow of each.

Examining the body of the while loop, I see that it basically increments `length` every iteration. It breaks out of the loop if the current character is not a digit, and applies some special case updates if the first character is an operator and the parser knows the input contains one. I can simplify the special casing using explanatory variables and use REPLACE NESTED CONDITIONALS WITH GUARD CLAUSES to reduce the number of nested scopes. I also changed the previously extracted method `isPastBoundaryOrNotDigit` - -which by its very name does more than 1 thing– into 2 separate methods that are invoked by the caller.

Figure C.14: insert caption

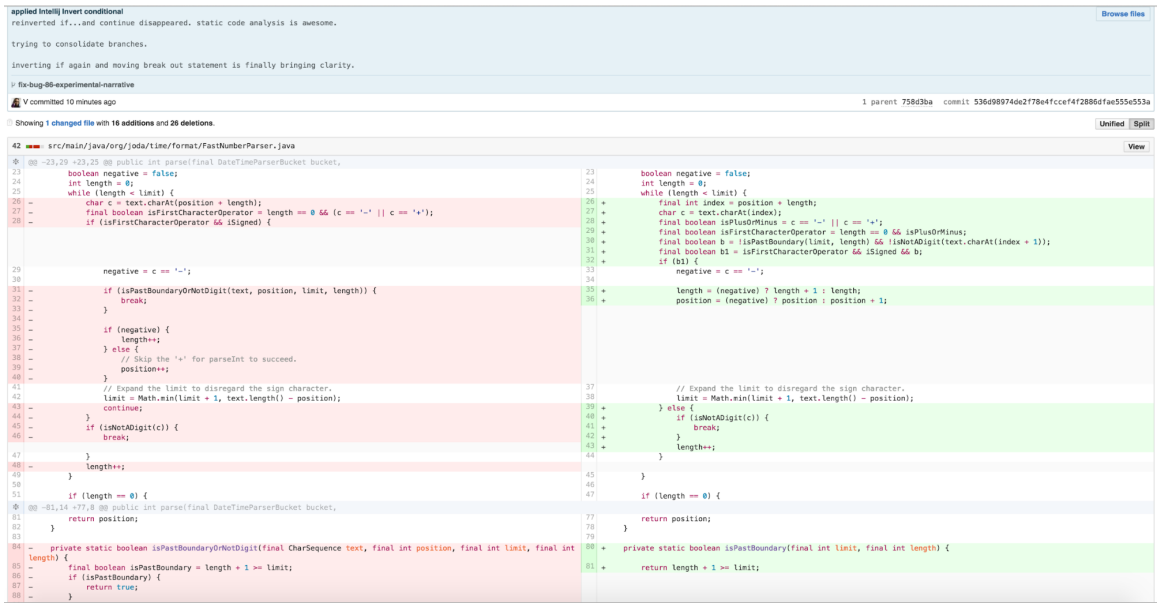


Figure C.15: insert caption

Seeking to further simplify the remaining `if/else` block inside the body of the `while`, I looked carefully at the special case logic for an operator and the `else` body and determined that both code paths incremented `length`. This type of conditional check to do the same thing in both cases is clearly extraneous cognitive load, so I applied a variant of CONSOLIDATE CONDITIONAL EXPRESSION.

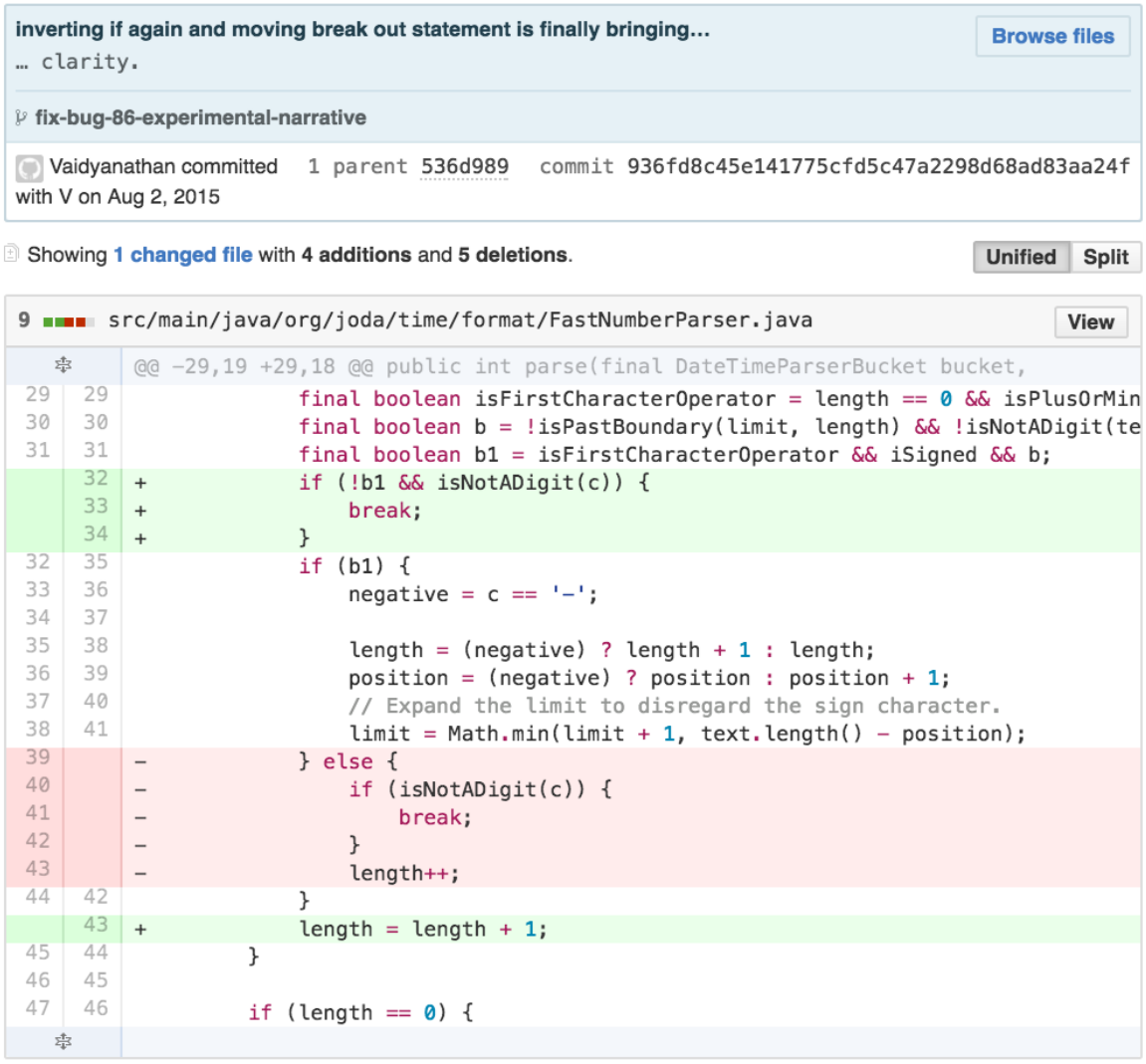


Figure C.16: insert caption

I then had an opportunity to simplify the while loop further by applying a similar procedure to a variant of REMOVE CONTROL FLAG to remove the break statement and clarify the loop predicate.

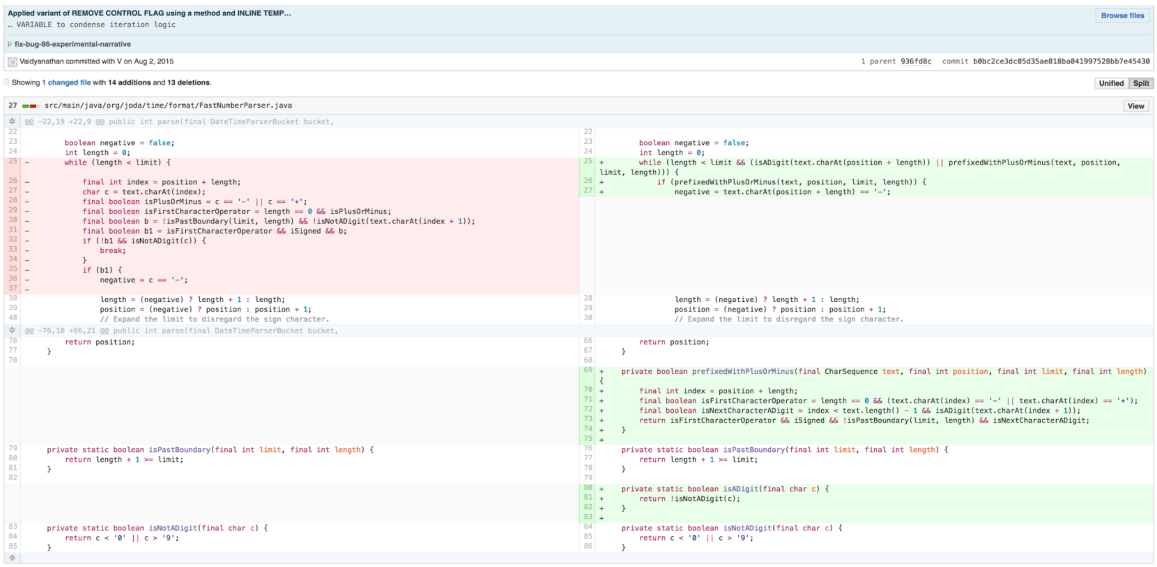


Figure C.17: insert caption

With this logic greatly simplified, I'm now able to push the calculation into a smaller, more cohesive chunk of logic using REPLACE METHOD WITH METHOD OBJECT.

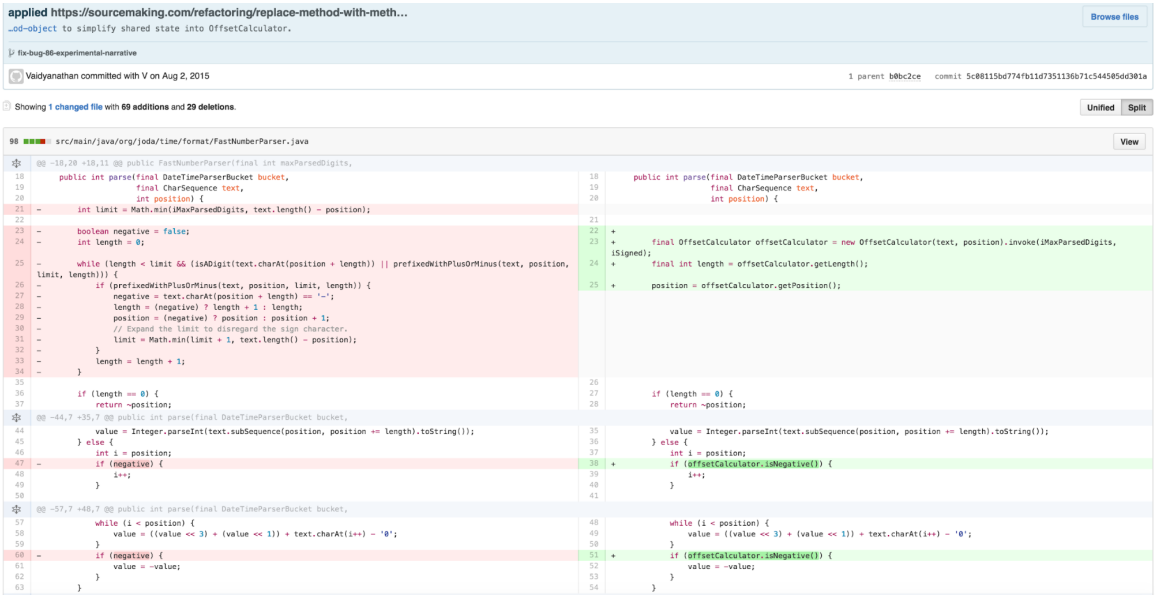


Figure C.18: insert caption

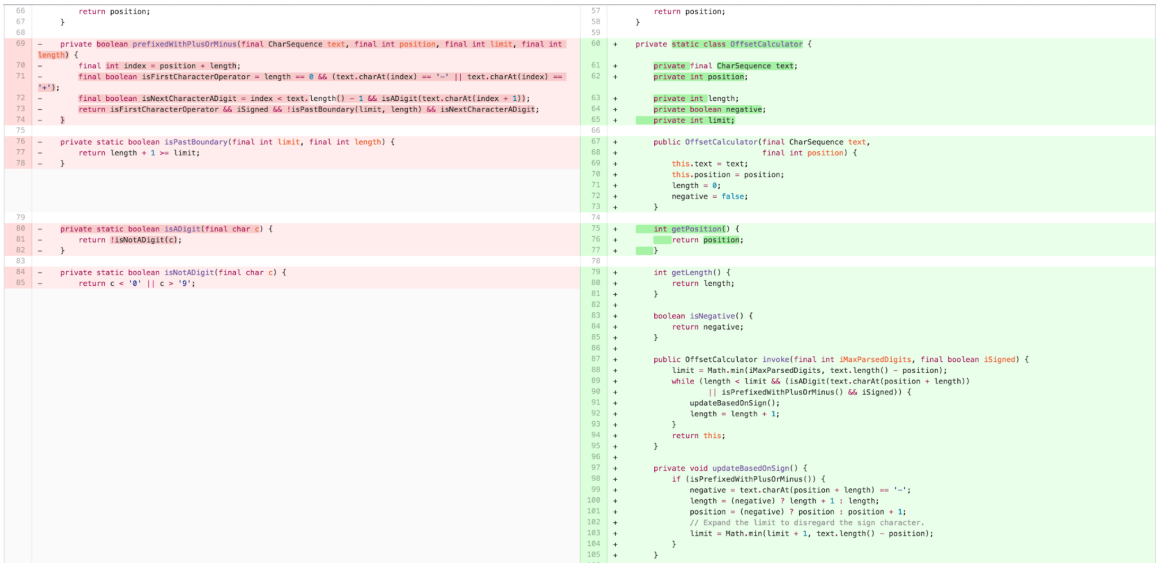


Figure C.19: insert caption

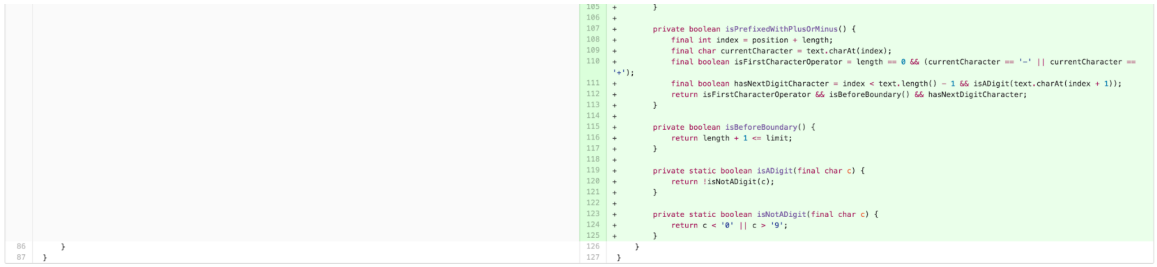


Figure C.20: insert caption

Next I wanted to ensure that the new class was inline with CLT's guidance on sequencing content effectively for comprehension, in line with the recommendations of Clean Code's STEPDOWN RULE/The Newspaper Metaphor.

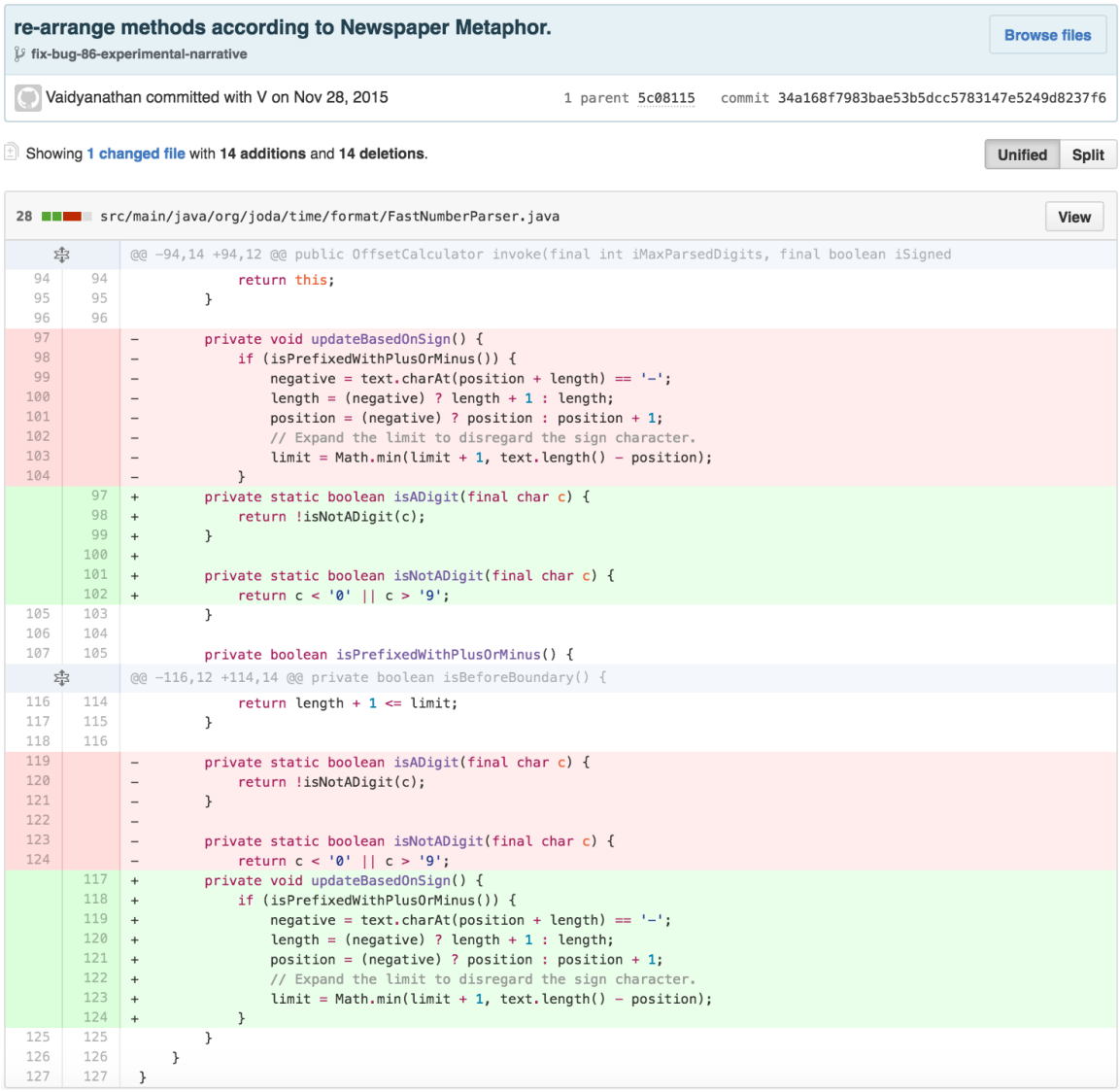


Figure C.21: insert caption

The new structure of the parse code had successfully dealt with the length and position calculation loop, but still had 35 lines. Much of the next body of work was involved in calculating a value with the stored computed values of `OffsetCalculator`. While this type of code practice is common, and the resulting `OffsetCalculator` is arguably much simpler, it violates the Tell, Don't Ask principle of object-oriented programming. Tell, Don't Ask helps manage the intrinsic cognitive load of systems by pushing operations on data closer to the data, making code more declarative. Following this approach, we move the value calculation into the `OffsetCalculator` and eliminate `FastNumberParser`.

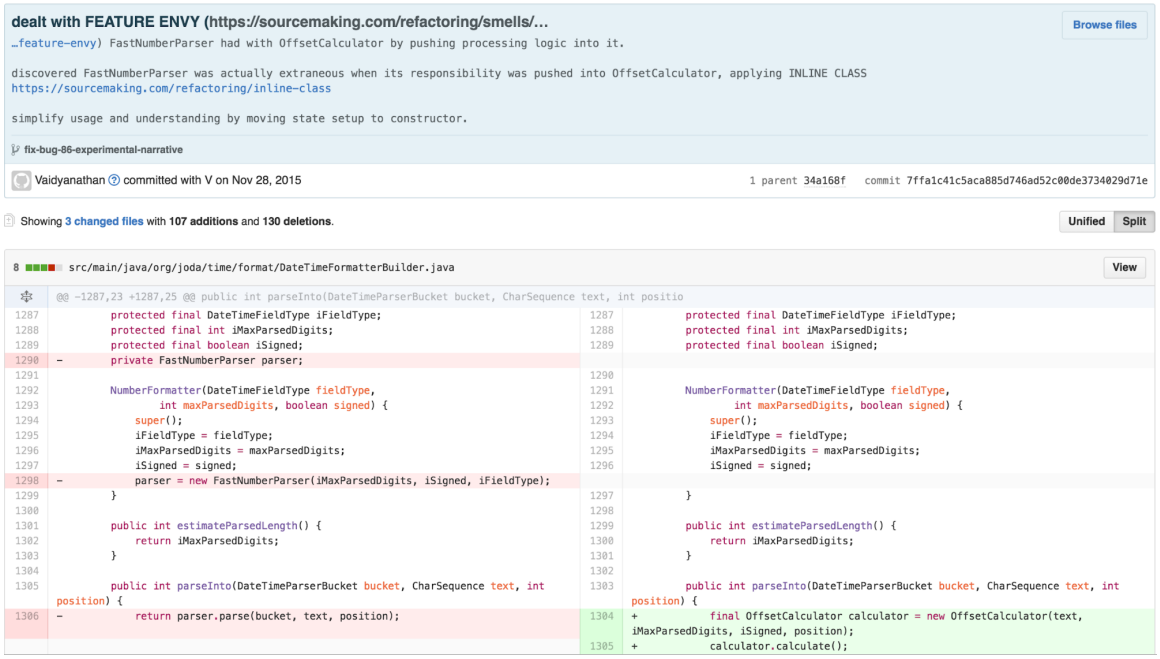


Figure C.22: insert caption

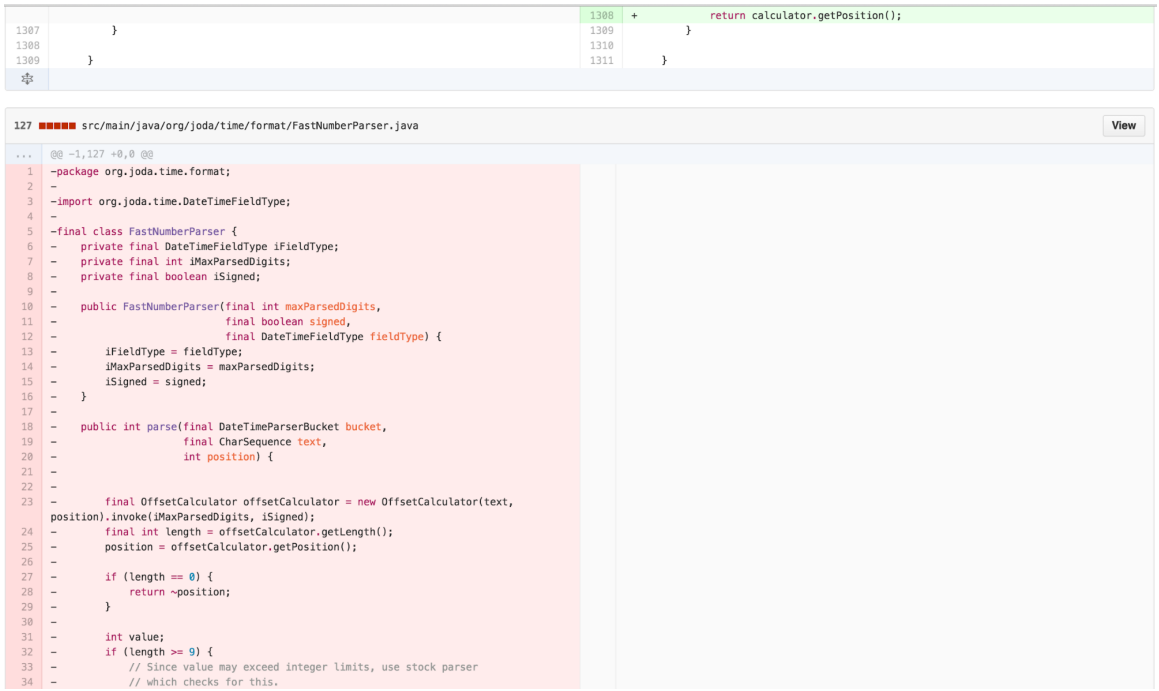
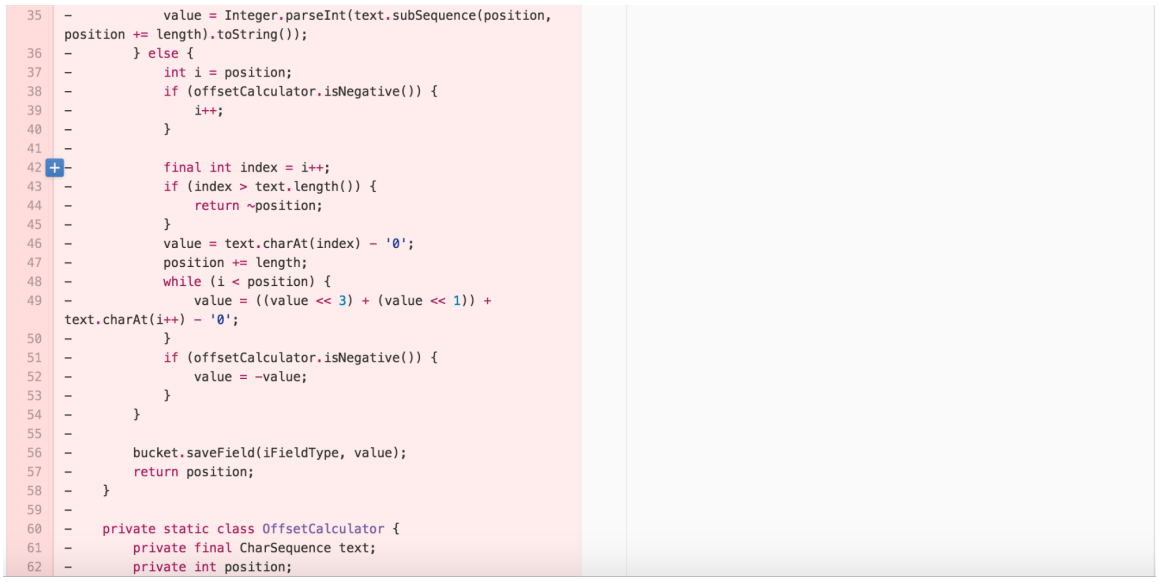
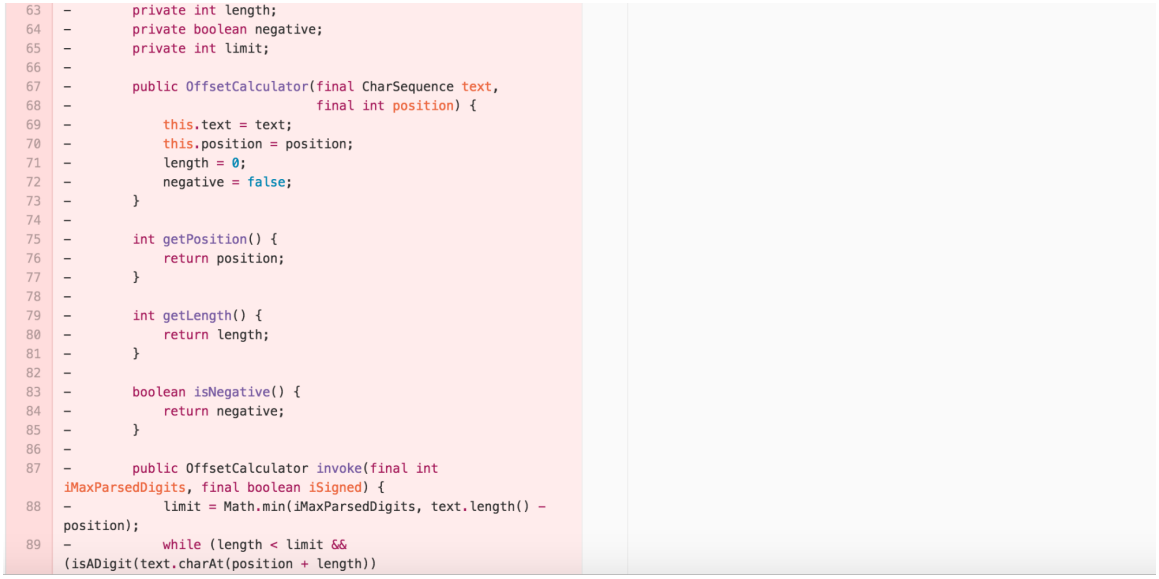


Figure C.23: insert caption



```
35  -            value = Integer.parseInt(text.subSequence(position,
    position += length).toString());
36  -        } else {
37  -            int i = position;
38  -            if (offsetCalculator.isNegative()) {
39  -                i++;
40  -            }
41  -
42  -            final int index = i++;
43  -            if (index > text.length()) {
44  -                return ~position;
45  -            }
46  -            value = text.charAt(index) - '0';
47  -            position += length;
48  -            while (i < position) {
49  -                value = ((value << 3) + (value << 1)) +
    text.charAt(i++) - '0';
50  -            }
51  -            if (offsetCalculator.isNegative()) {
52  -                value = -value;
53  -            }
54  -        }
55  -
56  -        bucket.saveField(iFieldType, value);
57  -        return position;
58  -    }
59  -
60  -    private static class OffsetCalculator {
61  -        private final CharSequence text;
62  -        private int position;
```

Figure C.24: insert caption

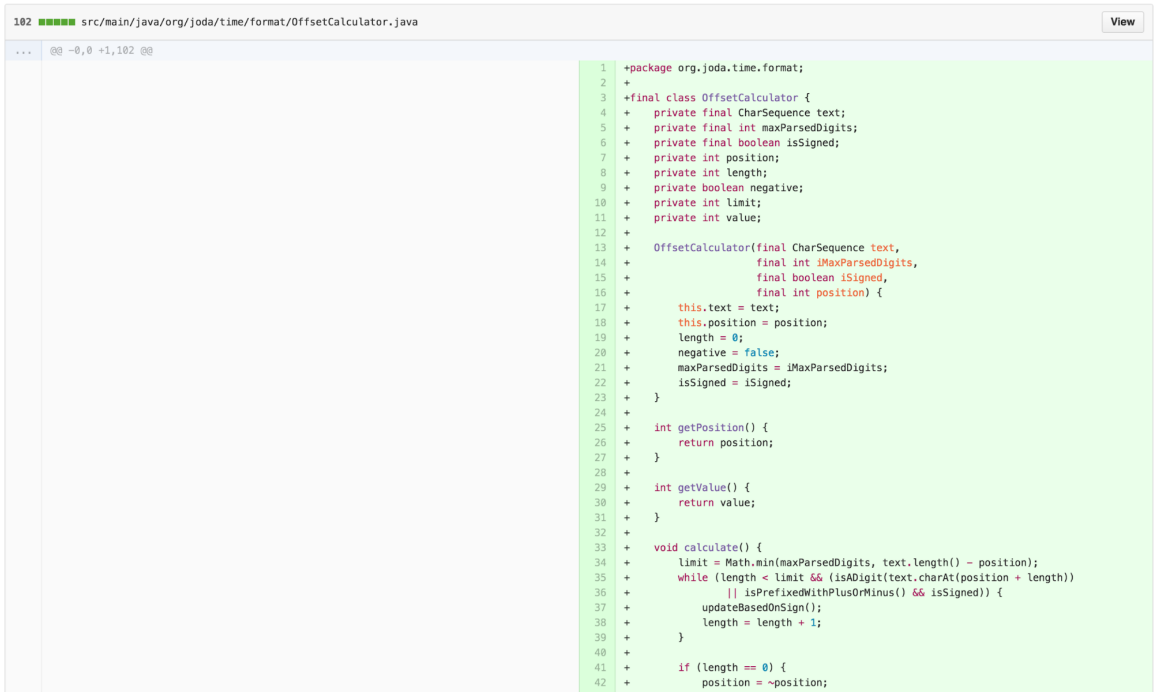


```
63  -        private int length;
64  -        private boolean negative;
65  -        private int limit;
66  -
67  -        public OffsetCalculator(final CharSequence text,
68  -                                final int position) {
69  -            this.text = text;
70  -            this.position = position;
71  -            length = 0;
72  -            negative = false;
73  -        }
74  -
75  -        int getPosition() {
76  -            return position;
77  -        }
78  -
79  -        int getLength() {
80  -            return length;
81  -        }
82  -
83  -        boolean isNegative() {
84  -            return negative;
85  -        }
86  -
87  -        public OffsetCalculator invoke(final int
    iMaxParsedDigits, final boolean iSigned) {
88  -            limit = Math.min(iMaxParsedDigits, text.length() -
    position);
89  -            while (length < limit &&
    (isADigit(text.charAt(position + length))
```

Figure C.25: insert caption

```
-                   || isPrefixedWithPlusOrMinus() && iSigned)) {
-               updateBasedOnSign();
-               length = length + 1;
-           }
-           return this;
-       }
-
-       private static boolean isADigit(final char c) {
-           return !isNotADigit(c);
-       }
-
-       private static boolean isNotADigit(final char c) {
-           return c < '0' || c > '9';
-       }
-
-       private boolean isPrefixedWithPlusOrMinus() {
-           final int index = position + length;
-           final char currentCharacter = text.charAt(index);
-           final boolean isFirstCharacterOperator = length == 0 && (currentCharacter
== '-' || currentCharacter == '+');
-           final boolean hasNextDigitCharacter = index < text.length() - 1 &&
isADigit(text.charAt(index + 1));
-           return isFirstCharacterOperator && isBeforeBoundary() &&
hasNextDigitCharacter;
-       }
-
-       private boolean isBeforeBoundary() {
-           return length + 1 <= limit;
-       }
-
-       private void updateBasedOnSign() {
-           if (isPrefixedWithPlusOrMinus()) {
-               negative = text.charAt(position + length) == '-';
-               length = (negative) ? length + 1 : length;
-               position = (negative) ? position : position + 1;
-               // Expand the limit to disregard the sign character.
-               limit = Math.min(limit + 1, text.length() - position);
-           }
-       }
-   }
-}
```

Figure C.26: insert caption



```
102 ■■■■■ src/main/java/org/joda/time/format/OffsetCalculator.java                View
...  @@ -0,0 +1,102 @@
+package org.joda.time.format;
+
+final class OffsetCalculator {
+    private final CharSequence text;
+    private final int maxParsedDigits;
+    private final boolean isSigned;
+    private int position;
+    private int length;
+    private boolean negative;
+    private int limit;
+    private int value;
+
+    OffsetCalculator(final CharSequence text,
+                     final int iMaxParsedDigits,
+                     final boolean iSigned,
+                     final int position) {
+        this.text = text;
+        this.position = position;
+        length = 0;
+        negative = false;
+        maxParsedDigits = iMaxParsedDigits;
+        isSigned = iSigned;
+    }
+
+    int getPosition() {
+        return position;
+    }
+
+    int getValue() {
+        return value;
+    }
+
+    void calculate() {
+        limit = Math.min(maxParsedDigits, text.length() - position);
+        while (length < limit && (isADigit(text.charAt(position + length))
+                || isPrefixedWithPlusOrMinus() && isSigned)) {
+            updateBasedOnSign();
+            length = length + 1;
+        }
+
+        if (length == 0) {
+            position = ~position;
```

Figure C.27: insert caption

```
+            return;
+        }
+
+        if (length >= 9) {
+            // Since value may exceed integer limits, use stock parser
+            // which checks for this.
+            value = Integer.parseInt(text.subSequence(position, position +=
length).toString());
+        } else {
+            int i = position;
+            if (negative) {
+                i++;
+            }
+
+            final int index = i++;
+            if (index > text.length()) {
+                position = ~position;
+                return;
+            }
+            value = text.charAt(index) - '0';
+            position += length;
+            while (i < position) {
+                value = ((value << 3) + (value << 1)) + text.charAt(i++) - '0';
+            }
+            if (negative) {
+                value = -value;
+            }
+        }
+
+    }
+
+    private static boolean isADigit(final char c) {
+        return !isNotADigit(c);
+    }
+
+    private static boolean isNotADigit(final char c) {
+        return c < '0' || c > '9';
+    }
+
+    private boolean isPrefixedWithPlusOrMinus() {
+        final int index = position + length;
+        final char currentCharacter = text.charAt(index);
+        final boolean isFirstCharacterOperator = length == 0 && (currentCharacter ==
'-' || currentCharacter == '+');
+        final boolean hasNextDigitCharacter = index < text.length() - 1 &&
isADigit(text.charAt(index + 1));
```

Figure C.28: insert caption

```
+        return isFirstCharacterOperator && isBeforeBoundary() &&
hasNextDigitCharacter;
+    }
+
+    private boolean isBeforeBoundary() {
+        return length + 1 <= limit;
+    }
+
+    private void updateBasedOnSign() {
+        if (isPrefixedWithPlusOrMinus()) {
+            negative = text.charAt(position + length) == '-';
+            length = (negative) ? length + 1 : length;
+            position = (negative) ? position : position + 1;
+            // Expand the limit to disregard the sign character.
+            limit = Math.min(limit + 1, text.length() -
position);
+        }
+    }
+}
```

Figure C.29: insert caption

We can simplify the body of calculate further by applying EXTRACT METHOD, INTRODUCE EXPLAINING VARIABLE, and applying One Return Per Function, an arguably easier-to-understand practice stemming from the principles of Structured Programming, to simplify control.

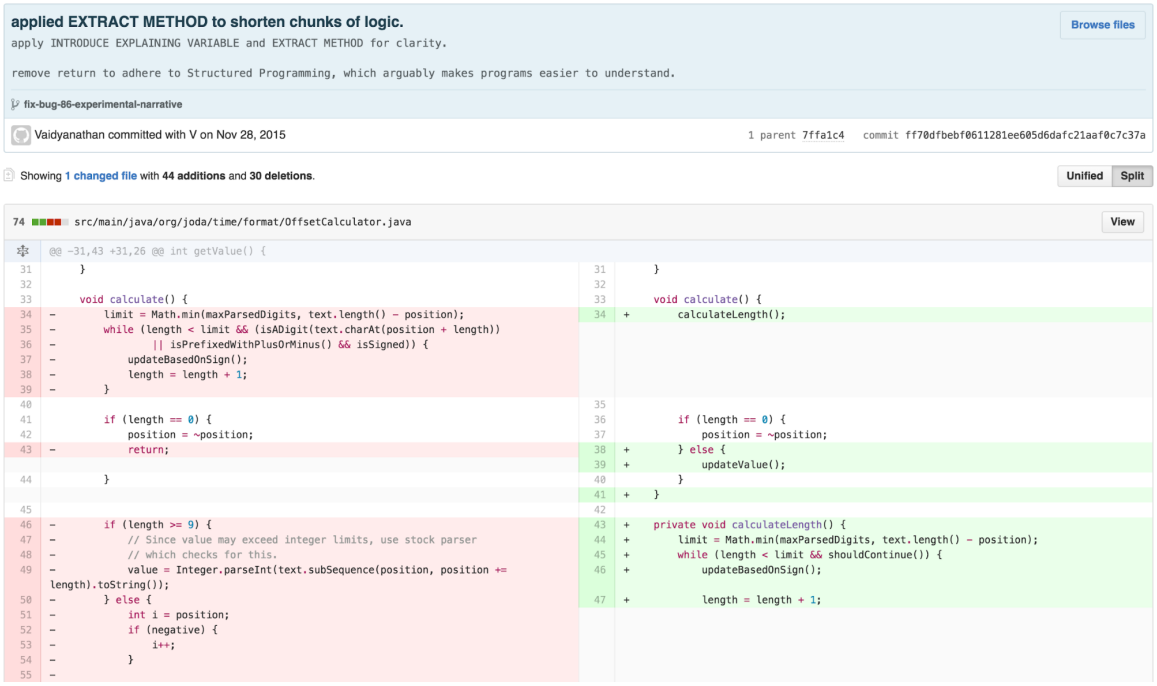
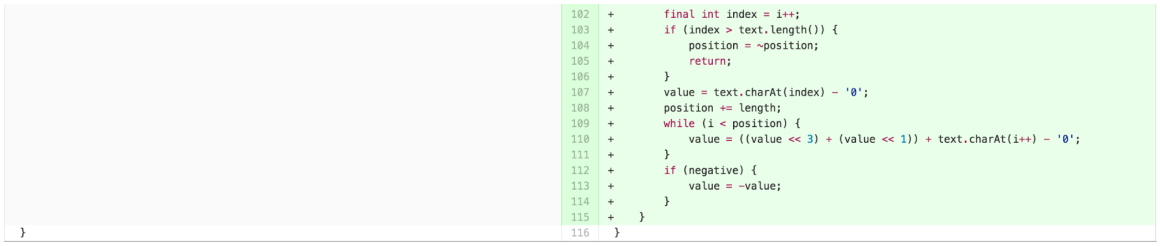


Figure C.30: insert caption

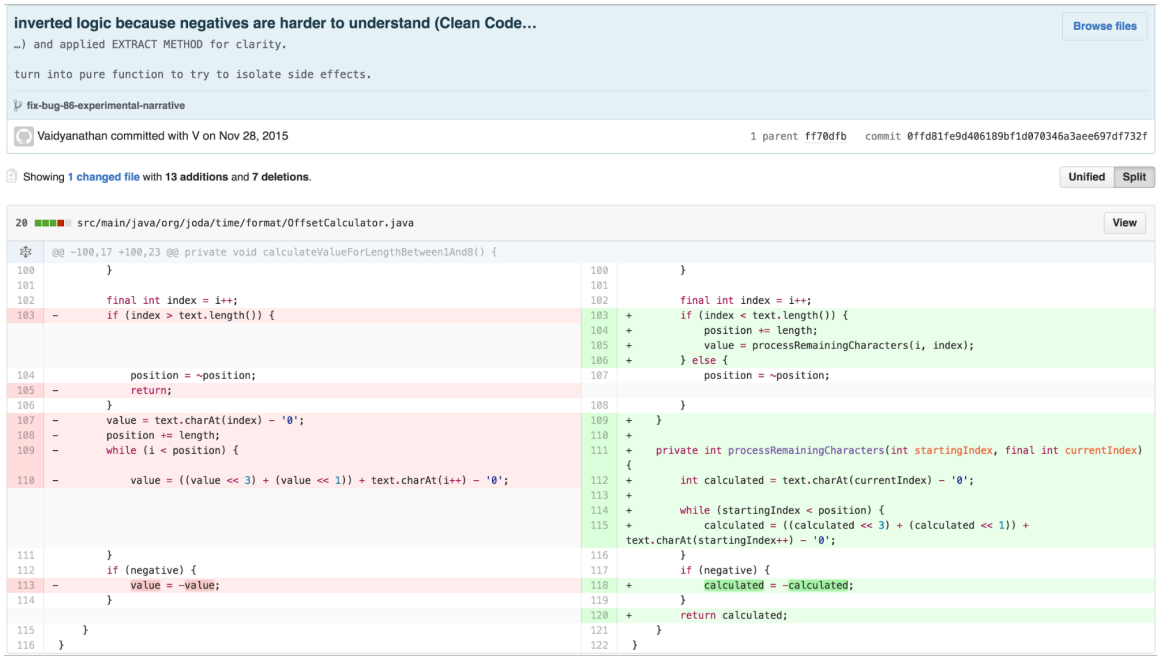
Figure C.31: insert caption



```
                                                    +        final int index = i++;
                                                    +        if (index > text.length()) {
                                                    +            position = ~position;
                                                    +            return;
                                                    +        }
                                                    +        value = text.charAt(index) - '0';
                                                    +        position += length;
                                                    +        while (i < position) {
                                                    +            value = ((value << 3) + (value << 1)) + text.charAt(i++) - '0';
                                                    +        }
                                                    +        if (negative) {
                                                    +            value = -value;
                                                    +        }
                                                    +    }
}                                                   }
```

Figure C.32: insert caption

I then simplified the logic of `calculateValueForLengthBetween1And8` by applying EXTRACT METHOD and inverting the logic of the conditional, as Clean Code suggests negatives are harder to understand and the inversion allowed me to remove a return statement that broke control flow.



inverted logic because negatives are harder to understand (Clean Code... Browse files

…) and applied EXTRACT METHOD for clarity.

turn into pure function to try to isolate side effects.

fix-bug-86-experimental-narrative

Vaidyanathan committed with V on Nov 28, 2015 — 1 parent ff70dfb commit 0ffd81fe9d406189bf1d070346a3aee697df732f

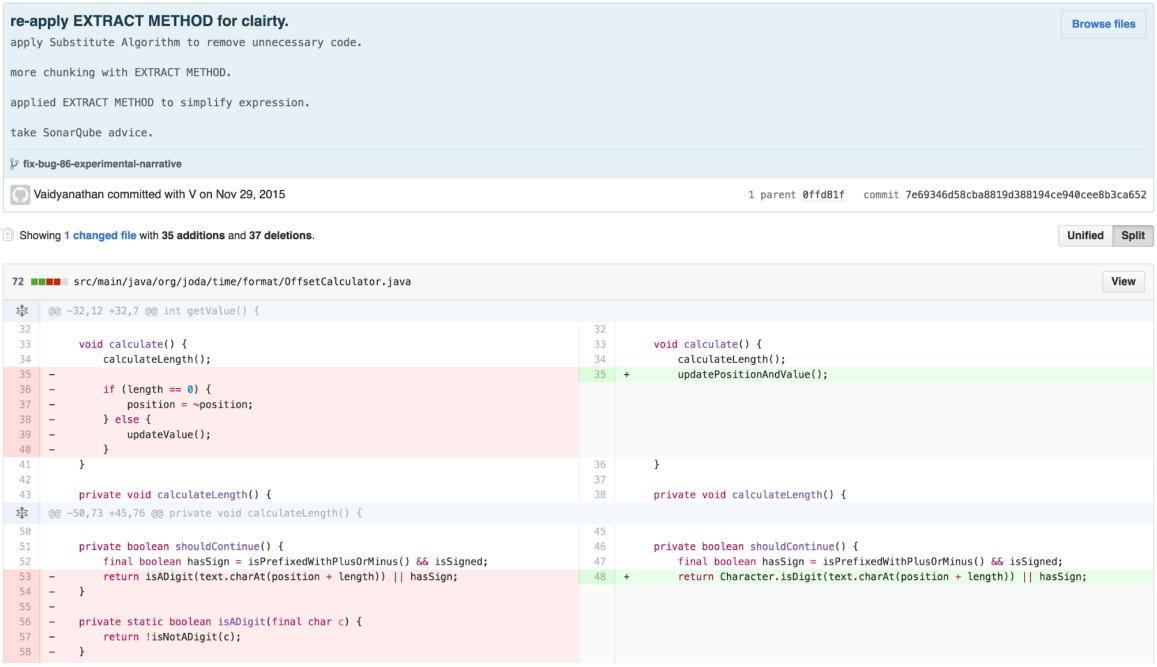
Showing 1 changed file with 13 additions and 7 deletions. Unified Split

20 src/main/java/org/joda/time/format/OffsetCalculator.java View

```
@@ -100,17 +100,23 @@ private void calculateValueForLengthBetween1And8() {
        }                                                   }

        final int index = i++;                              final int index = i++;
-       if (index > text.length()) {                        +       if (index < text.length()) {
                                                            +           position += length;
                                                            +           value = processRemainingCharacters(i, index);
                                                            +       } else {
            position = ~position;                               position = ~position;
-           return;
        }                                                   }
-       value = text.charAt(index) - '0';                   +   }
-       position += length;                                 +
-       while (i < position) {                              +   private int processRemainingCharacters(int startingIndex, final int currentIndex)
                                                                {
-           value = ((value << 3) + (value << 1)) + text.charAt(i++) - '0';   +       int calculated = text.charAt(currentIndex) - '0';
                                                            +
                                                            +       while (startingIndex < position) {
                                                            +           calculated = ((calculated << 3) + (calculated << 1)) +
                                                            text.charAt(startingIndex++) - '0';
        }                                                   }
        if (negative) {                                     if (negative) {
-           value = -value;                                 +           calculated = -calculated;
        }                                                   }
                                                            +       return calculated;
    }                                                       }
}                                                       }
```

Figure C.33: insert caption

I then started seeking code to remove, as I detected that some of the methods were redundant from features already offered by the built-in Java libraries. I also applied further EXTRACT METHOD to describe the algorithm declaratively using names and separate the levels of abstraction according to the Newspaper Metaphor.



re-apply EXTRACT METHOD for clairty.

Browse files

apply Substitute Algorithm to remove unnecessary code.

more chunking with EXTRACT METHOD.

applied EXTRACT METHOD to simplify expression.

take SonarQube advice.

fix-bug-86-experimental-narrative

Vaidyanathan committed with V on Nov 29, 2015 — 1 parent 0ffd81f commit 7e69346d58cba8819d388194ce940cee8b3ca652

Showing 1 changed file with 35 additions and 37 deletions. Unified Split

72 src/main/java/org/joda/time/format/OffsetCalculator.java View

```
@@ -32,12 +32,7 @@ int getValue() {
     void calculate() {
         calculateLength();
-
-        if (length == 0) {
-            position = ~position;
-        } else {
-            updateValue();
-        }
+        updatePositionAndValue();
     }

     private void calculateLength() {
@@ -50,73 +45,76 @@ private void calculateLength() {

     private boolean shouldContinue() {
         final boolean hasSign = isPrefixedWithPlusOrMinus() && isSigned;
-        return isADigit(text.charAt(position + length)) || hasSign;
+        return Character.isDigit(text.charAt(position + length)) || hasSign;
     }

-    private static boolean isADigit(final char c) {
-        return !isNotADigit(c);
-    }
```

Figure C.34: insert caption

```
-    private static boolean isNotADigit(final char c) {
-        return c < '0' || c > '9';
     }

     private boolean isPrefixedWithPlusOrMinus() {
         final int index = position + length;
         final char currentCharacter = text.charAt(index);
-        final boolean isFirstCharacterOperator = length == 0 && (currentCharacter ==
'-' || currentCharacter == '+');
-        final boolean hasNextDigitCharacter = index < text.length() - 1 &&
isADigit(text.charAt(index + 1));
+        final boolean isFirstCharacterOperator = length == 0 &&
isCharacterOperator(currentCharacter);
+        final boolean hasNextDigitCharacter = index < text.length() - 1 &&
Character.isDigit(text.charAt(index + 1));
         return isFirstCharacterOperator && isBeforeBoundary() &&
hasNextDigitCharacter;
     }

+    private static boolean isCharacterOperator(final char currentCharacter) {
+        return currentCharacter == '-' || currentCharacter == '+';
+    }
+

     private boolean isBeforeBoundary() {
         return length + 1 <= limit;
     }

     private void updateBasedOnSign() {
         if (isPrefixedWithPlusOrMinus()) {
             negative = text.charAt(position + length) == '-';
-            length = (negative) ? length + 1 : length;
-            position = (negative) ? position : position + 1;
+            length = negative ? length + 1 : length;
+            position = negative ? position : position + 1;
             // Expand the limit to disregard the sign character.
             limit = Math.min(limit + 1, text.length() - position);
         }
     }

-    private void updateValue() {
-        if (length >= 9) {
-            // Since value may exceed integer limits, use stock parser
-            // which checks for this.
-            value = Integer.parseInt(text.subSequence(position, position +=
length).toString());
+    private void updatePositionAndValue() {
+        if (length == 0) {
+            position = ~position;
+        } else if (length >= 9) {
+            useDefaultParser();
         } else {
-            calculateValueForLengthBetween1And8();
+            useFastParser();
         }
     }
```

Figure C.35: insert caption

Figure C.36: insert caption

At this point, from a pure "code metrics" perspective, `OffsetCalculator` was significantly simpler. Each method was no more than 9 lines long, the class had 3 getter methods, 3 "utility" style 1-line descriptive methods, and 8 "logical" methods. Being pretty experienced with the code at this point, it was fairly easy for me to see where to "apply the fix" to resolve the bug in this reduced scope.

C.6.1 Checking my biases: adapting to peer feedback

I was pretty confident the new code would be significantly easier to understand. In order to validate this hypothesis, I sought feedback from some expert and novice software engineers. In these dry-run, non-formal talk throughs, I discovered something surprising. Although the `OffsetCalculator` was chunked well for comprehensibility, its overall role within the parsing process was difficult to understand. The obviously tricky `useFastParser()` that used bit shifting was cognitively dense, but even the calculation of length and position that immediately preceded it were difficult to grok in terms of the intent of the calculation. Initial passes at debugging showed that the architecture itself, from the relationship of the `DateTimeFormatter` to the `DateTimeFormatterBuilder` to the `NumberFormatter` to the `OffsetCalculator`, was complex. I realized I need to spend more time with the `DateTimeFormatter` to make it easier to bypass its functionality and get to the `OffsetCalculator` quicker.

C.6.2 Architectural adaptation: Simplifying DateTimeFormatter

I started with applying EXTRACT METHOD on spots of obvious duplication the calculation of Chronology, into a method called `getChronology`. I then saw repeated duplication with only detail variation in `parseDateTime`, `parseLocalDateTime`, and `parseMutableDateTime`, so I extracted what was different in those methods out. In

order to create a seam for myself to be able to move the body of those methods into an independently simplified piece without changing the externally facing interface, I extracted the body of the public methods into separate internal private methods.

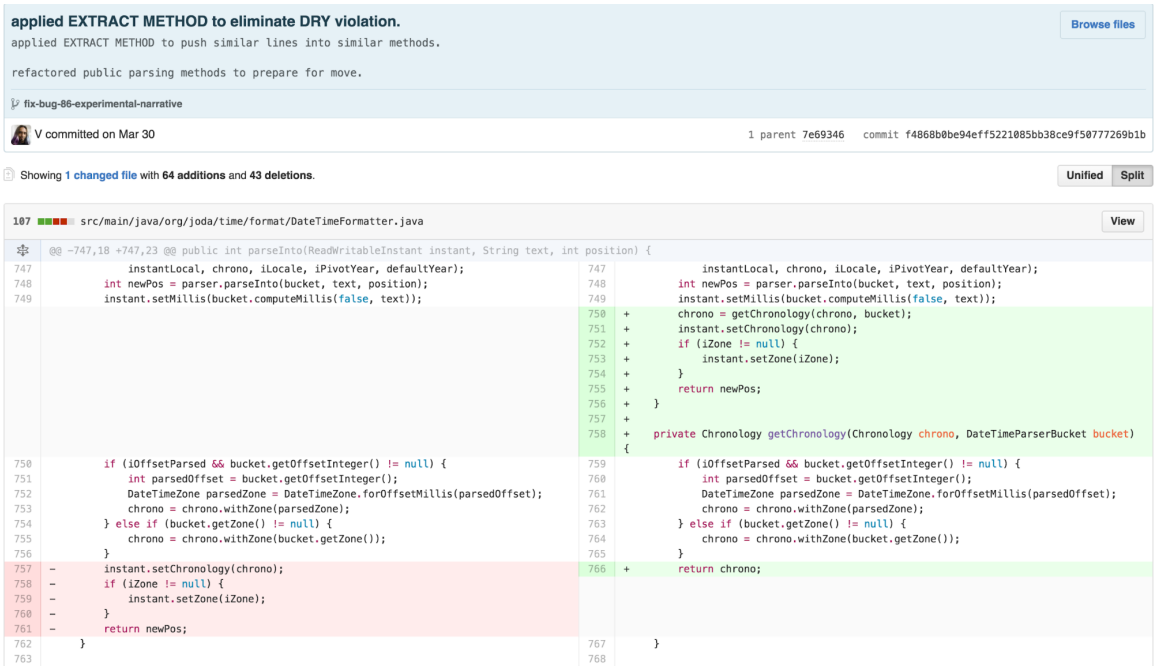


Figure C.37: insert caption

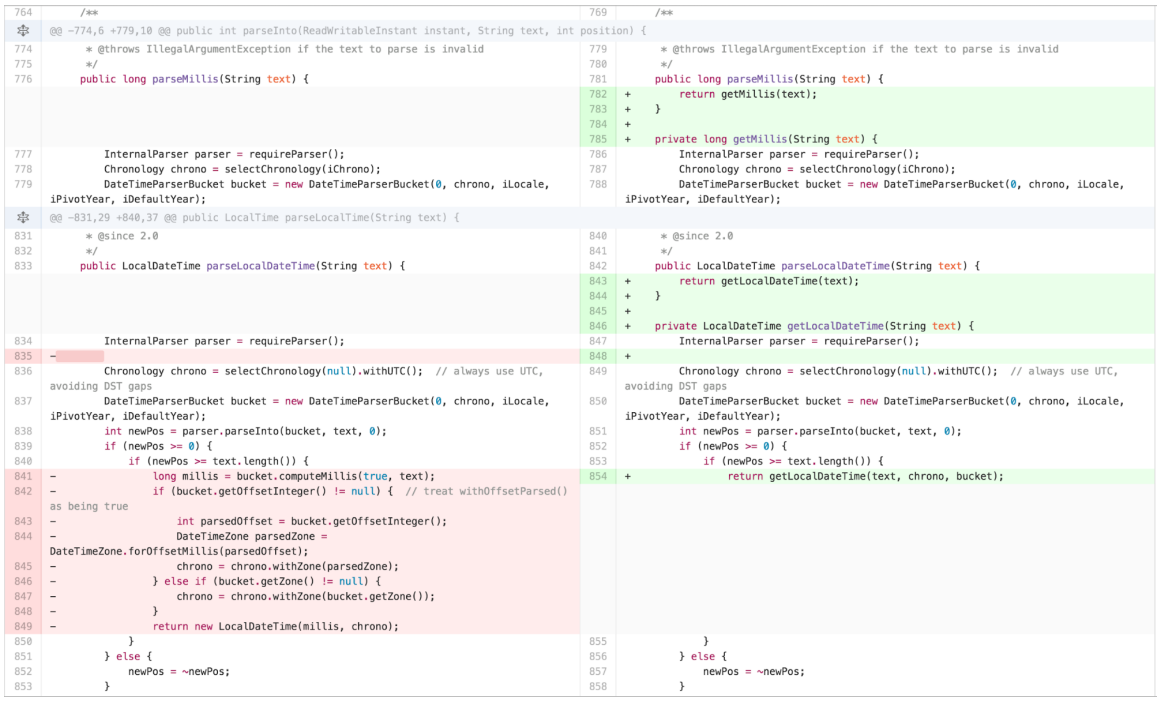


Figure C.38: insert caption

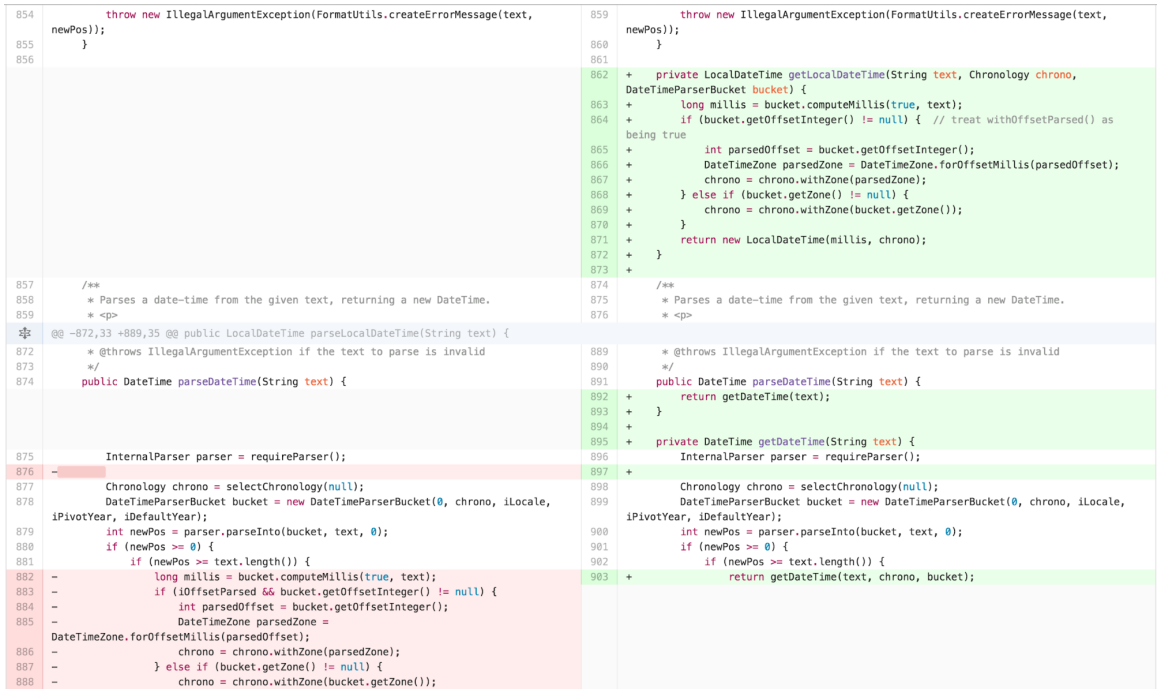


Figure C.39: insert caption

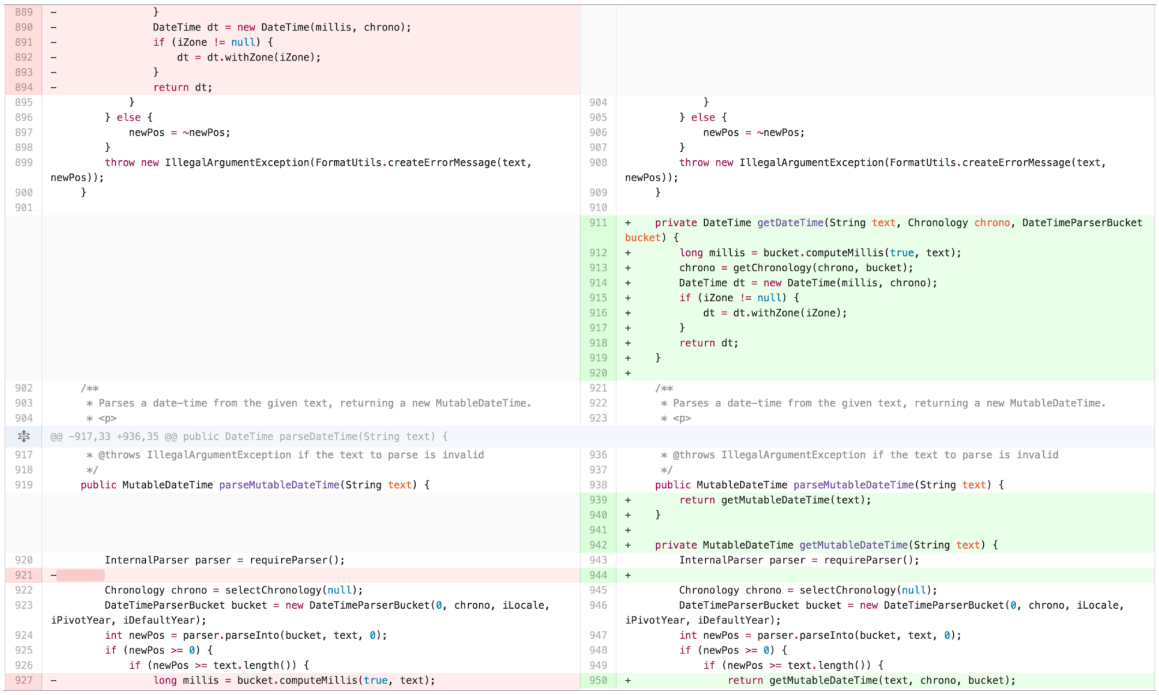


Figure C.40: insert caption

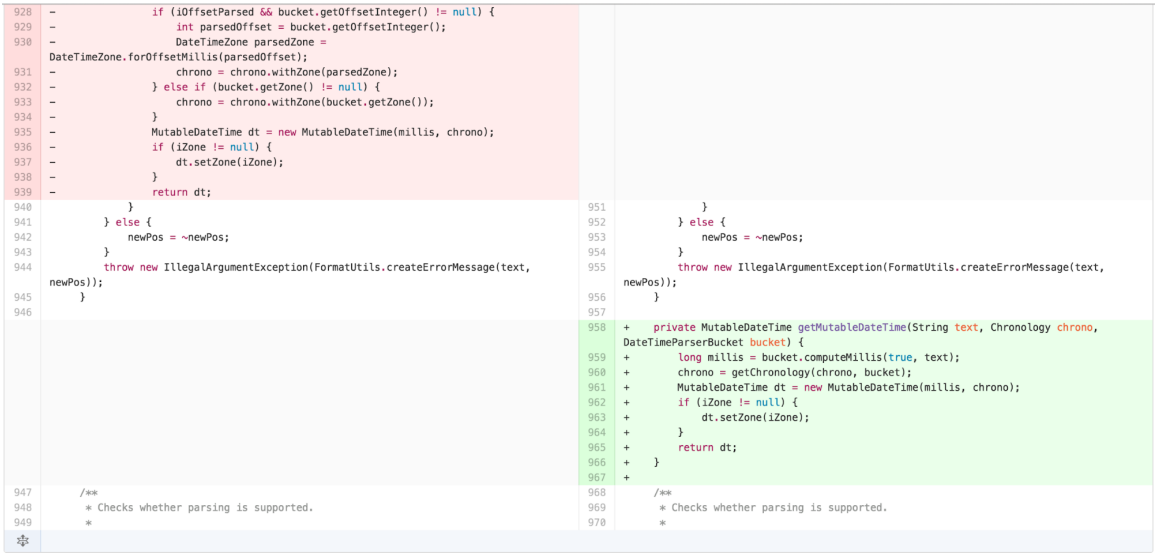


Figure C.41: insert caption

I then pushed the responsibility for `Chronology` calculation into a `ChronologyFactory`.

EXTRACT METHOD for Chronology functionality.

Browse files

push Chronology selection functionality to ChronologyFactory.

fix-bug-86-experimental-narrative

V committed on Mar 30 — 1 parent f4868b0 commit 4a6c2077d2fa249fd49e07d8b3945c278b26a04b

Showing 2 changed files with 42 additions and 22 deletions. Unified Split

36 src/main/java/org/joda/time/format/ChronologyFactory.java View

```
@@ -0,0 +1,36 @@
+package org.joda.time.format;
+
+import org.joda.time.Chronology;
+import org.joda.time.DateTimeUtils;
+import org.joda.time.DateTimeZone;
+
+final class ChronologyFactory {
+    private ChronologyFactory() {}
+    /**
+     * Determines the correct chronology to use.
+     *
+     * @param defaultChronology
+     * @param defaultTZ         @param chrono  the proposed chronology
+     * @return the actual chronology
+     */
+    static Chronology selectChronology(Chronology defaultChronology, DateTimeZone defaultTZ, Chronology chrono) {
+        chrono = getChronologyWithDefaultValue(defaultChronology, chrono);
+        chrono = getChronologyWithTimeZone(chrono, defaultTZ);
+        return chrono;
+    }
+
+    static Chronology getChronologyWithDefaultValue(Chronology defaultChronology, Chronology chrono) {
+        chrono = DateTimeUtils.getChronology(chrono);
+        if (defaultChronology != null) {
+            chrono = defaultChronology;
+        }
+        return chrono;
+    }
+
```

Figure C.42: insert caption

```
+    static Chronology getChronologyWithTimeZone(Chronology chrono, DateTimeZone defaultTimeZone) {
+        if (defaultTimeZone != null) {
+            chrono = chrono.withZone(defaultTimeZone);
+        }
+        return chrono;
+    }
+}
```

28 src/main/java/org/joda/time/format/DateTimeFormatter.java View

```
@@ -669,7 +669,7 @@ public String print(ReadablePartial partial) {

     private void printTo(Appendable appendable, long instant, Chronology chrono) throws IOException {
         InternalPrinter printer = requirePrinter();
-        chrono = selectChronology(chrono);
+        chrono = ChronologyFactory.selectChronology(iChrono, iZone, chrono);
         // Shift instant into local time (UTC) to avoid excessive offset
         // calculations when printing multiple fields in a composite printer.
         DateTimeZone zone = chrono.getZone();
@@ -741,7 +741,7 @@ public int parseInto(ReadWritableInstant instant, String text, int position) {
         Chronology chrono = instant.getChronology();
         int defaultYear = DateTimeUtils.getChronology(chrono).year().get(instantMillis);
         long instantLocal = instantMillis + chrono.getZone().getOffset(instantMillis);
-        chrono = selectChronology(chrono);
+        chrono = ChronologyFactory.selectChronology(iChrono, iZone, chrono);

         DateTimeParserBucket bucket = new DateTimeParserBucket(
             instantLocal, chrono, iLocale, iPivotYear, defaultYear);
@@ -784,7 +784,7 @@ public long parseMillis(String text) {

     private long getMillis(String text) {
         InternalParser parser = requireParser();
-        Chronology chrono = selectChronology(iChrono);
+        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, iChrono);
         DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
         return bucket.doParseMillis(parser, text);
     }
@@ -846,7 +846,7 @@ public LocalDateTime parseLocalDateTime(String text) {
     private LocalDateTime getLocalDateTime(String text) {
         InternalParser parser = requireParser();

-        Chronology chrono = selectChronology(null).withUTC();  // always use UTC, avoiding DST gaps
+        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null).withUTC();  // always use UTC, avo
```

Figure C.43: insert caption

```
850  850          DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
851  851          int newPos = parser.parseInto(bucket, text, 0);
852  852          if (newPos >= 0) {
     @@ -895,7 +895,7 @@ public DateTime parseDateTime(String text) {
895  895      private DateTime getDateTime(String text) {
896  896          InternalParser parser = requireParser();
897  897
898       -          Chronology chrono = selectChronology(null);
          898  +          Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
899  899          DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
900  900          int newPos = parser.parseInto(bucket, text, 0);
901  901          if (newPos >= 0) {
     @@ -942,7 +942,7 @@ public MutableDateTime parseMutableDateTime(String text) {
942  942      private MutableDateTime getMutableDateTime(String text) {
943  943          InternalParser parser = requireParser();
944  944
945       -          Chronology chrono = selectChronology(null);
          945  +          Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
946  946          DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
947  947          int newPos = parser.parseInto(bucket, text, 0);
948  948          if (newPos >= 0) {
     @@ -979,21 +979,5 @@ private InternalParser requireParser() {
979  979      }
980  980
981  981      //-----------------------------------------------------------------------
982       -    /**
983       -     * Determines the correct chronology to use.
984       -     *
985       -     * @param chrono  the proposed chronology
986       -     * @return the actual chronology
987       -     */
988       -    private Chronology selectChronology(Chronology chrono) {
989       -        chrono = DateTimeUtils.getChronology(chrono);
990       -        if (iChrono != null) {
991       -            chrono = iChrono;
992       -        }
993       -        if (iZone != null) {
994       -            chrono = chrono.withZone(iZone);
995       -        }
996       -        return chrono;
997       -    }
998  982
999  983  }
```

Figure C.44: insert caption

I then saw an opportunity. The similarity of the bodies of the `parseDateTime`, `parseLocalDateTime`, and `parseMutableDateTime` methods suggested to me that a TEMPLATE METHOD was hidden in this class.

The use of a Design Pattern can increase the Germane Cognitive Load of an architecture, which can be more difficult for novices to understand. However, application of this pattern here likely resulted in a dramatically simpler `DateTimeFormatter` and better coherence in the architecture. Following Guideline 25: Write High Coherent Texts for Low Knowledge Readers, this strategic use of a design pattern seemed likely to make the code easier to understand for novices as well. Hence, I started working to make this pattern manifest. Through a series of commits, I realized that `DateTimeFormatter` had a lot of FEATURE ENVY with `DateTimeParserBucket`.

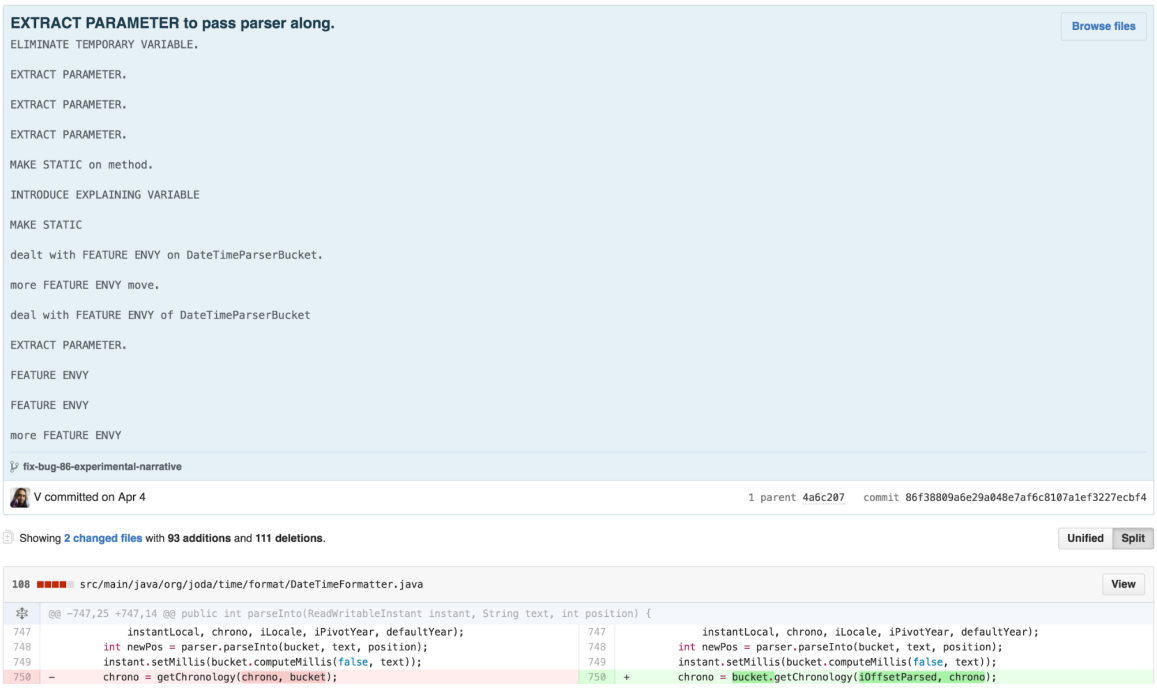


Figure C.45: insert caption

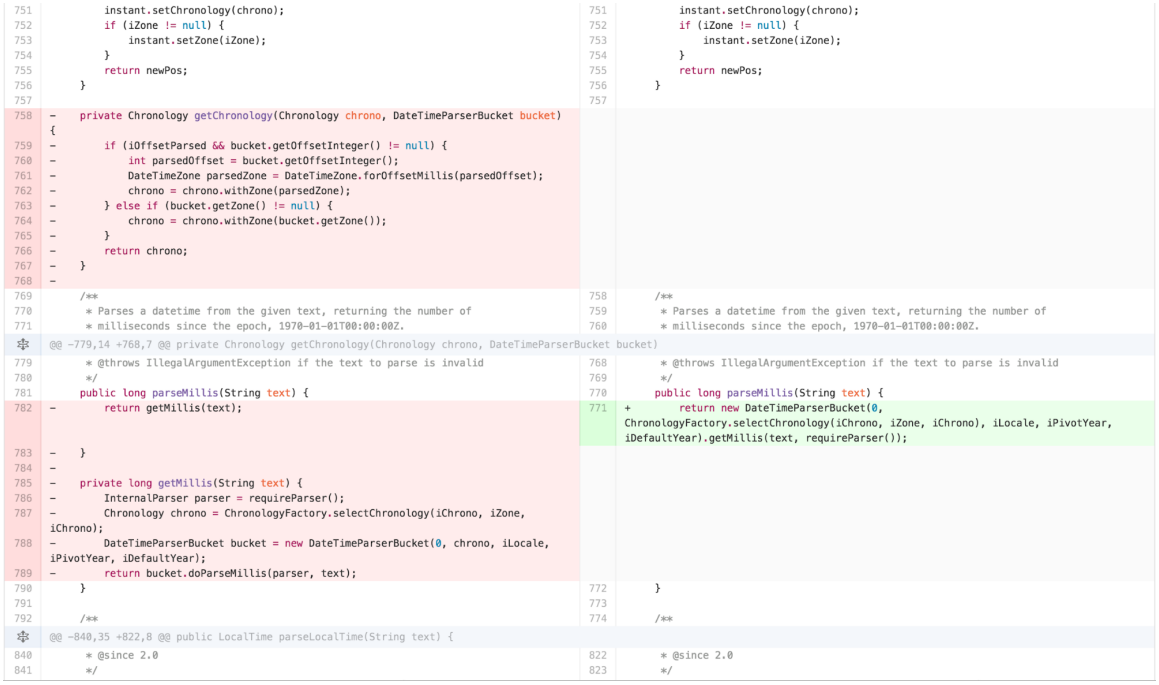
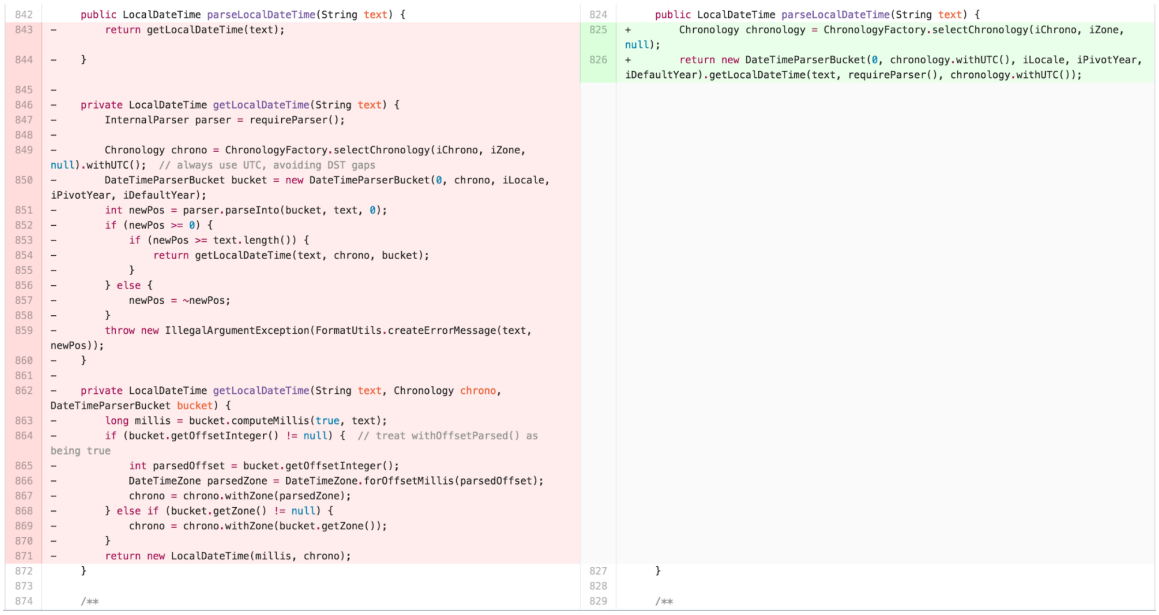


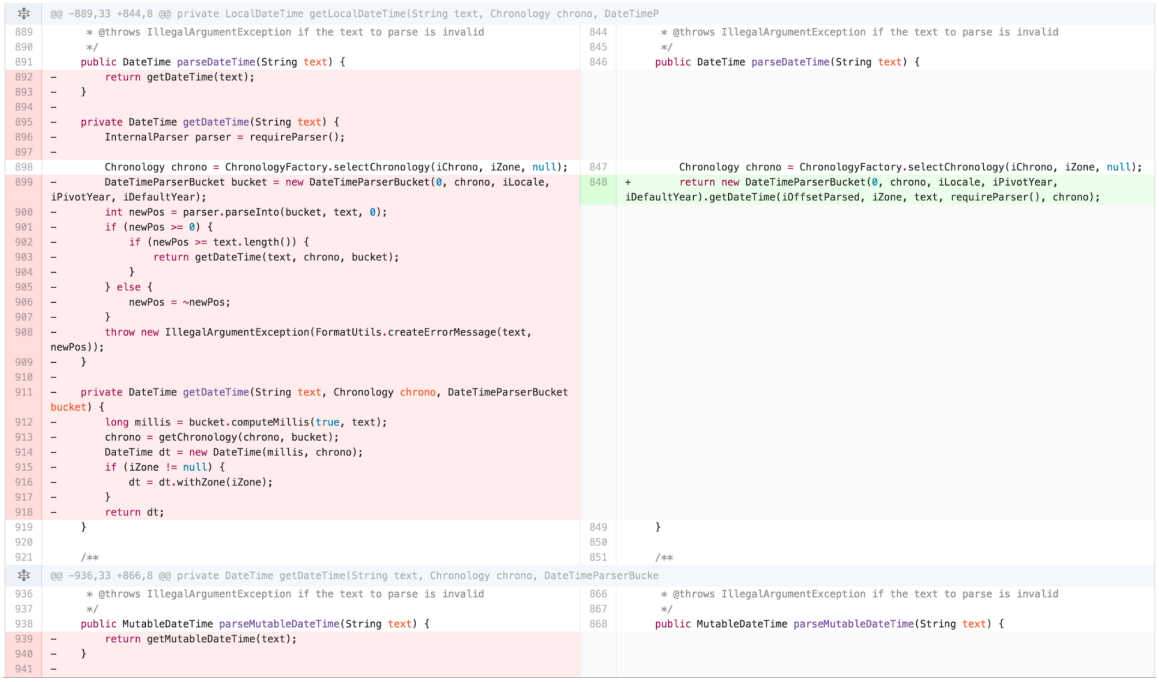
Figure C.46: insert caption



```
-    public LocalDateTime parseLocalDateTime(String text) {
-        return getLocalDateTime(text);
-    }
-
-    private LocalDateTime getLocalDateTime(String text) {
-        InternalParser parser = requireParser();
-
-        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null).withUTC();  // always use UTC, avoiding DST gaps
-        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
-        int newPos = parser.parseInto(bucket, text, 0);
-        if (newPos >= 0) {
-            if (newPos >= text.length()) {
-                return getLocalDateTime(text, chrono, bucket);
-            }
-        } else {
-            newPos = ~newPos;
-        }
-        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
-    }
-
-    private LocalDateTime getLocalDateTime(String text, Chronology chrono, DateTimeParserBucket bucket) {
-        long millis = bucket.computeMillis(true, text);
-        if (bucket.getOffsetInteger() != null) {  // treat withOffsetParsed() as being true
-            int parsedOffset = bucket.getOffsetInteger();
-            DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
-            chrono = chrono.withZone(parsedZone);
-        } else if (bucket.getZone() != null) {
-            chrono = chrono.withZone(bucket.getZone());
-        }
-        return new LocalDateTime(millis, chrono);
     }

     /**
```

```
     public LocalDateTime parseLocalDateTime(String text) {
+        Chronology chronology = ChronologyFactory.selectChronology(iChrono, iZone, null);
+        return new DateTimeParserBucket(0, chronology.withUTC(), iLocale, iPivotYear, iDefaultYear).getLocalDateTime(text, requireParser(), chronology.withUTC());
     }

     /**
```

Figure C.47: insert caption



```
@@ -889,33 +844,8 @@ private LocalDateTime getLocalDateTime(String text, Chronology chrono, DateTimeP
      * @throws IllegalArgumentException if the text to parse is invalid
      */
     public DateTime parseDateTime(String text) {
-        return getDateTime(text);
-    }
-
-    private DateTime getDateTime(String text) {
-        InternalParser parser = requireParser();
-
         Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
-        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
-        int newPos = parser.parseInto(bucket, text, 0);
-        if (newPos >= 0) {
-            if (newPos >= text.length()) {
-                return getDateTime(text, chrono, bucket);
-            }
-        } else {
-            newPos = ~newPos;
-        }
-        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
-    }
-
-    private DateTime getDateTime(String text, Chronology chrono, DateTimeParserBucket bucket) {
-        long millis = bucket.computeMillis(true, text);
-        chrono = getChronology(chrono, bucket);
-        DateTime dt = new DateTime(millis, chrono);
-        if (iZone != null) {
-            dt = dt.withZone(iZone);
-        }
-        return dt;
+        return new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear).getDateTime(iOffsetParsed, iZone, text, requireParser(), chrono);
     }

     /**
@@ -936,33 +866,8 @@ private DateTime getDateTime(String text, Chronology chrono, DateTimeParserBucke
      * @throws IllegalArgumentException if the text to parse is invalid
      */
     public MutableDateTime parseMutableDateTime(String text) {
-        return getMutableDateTime(text);
-    }
-
-    private MutableDateTime getMutableDateTime(String text) {
-        InternalParser parser = requireParser();
-
         Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
-        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
-        int newPos = parser.parseInto(bucket, text, 0);
-        if (newPos >= 0) {
-            if (newPos >= text.length()) {
-                return getMutableDateTime(text, chrono, bucket);
-            }
-        } else {
-            newPos = ~newPos;
-        }
-        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
-    }
-
-    private MutableDateTime getMutableDateTime(String text, Chronology chrono, DateTimeParserBucket bucket) {
-        long millis = bucket.computeMillis(true, text);
-        chrono = getChronology(chrono, bucket);
-        MutableDateTime dt = new MutableDateTime(millis, chrono);
-        if (iZone != null) {
-            dt = dt.withZone(iZone);
-        }
-        return dt;
+        return new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear).getMutableDateTime(iOffsetParsed, iZone, text, requireParser(), chrono);
     }

     /**
```

Figure C.48: insert caption

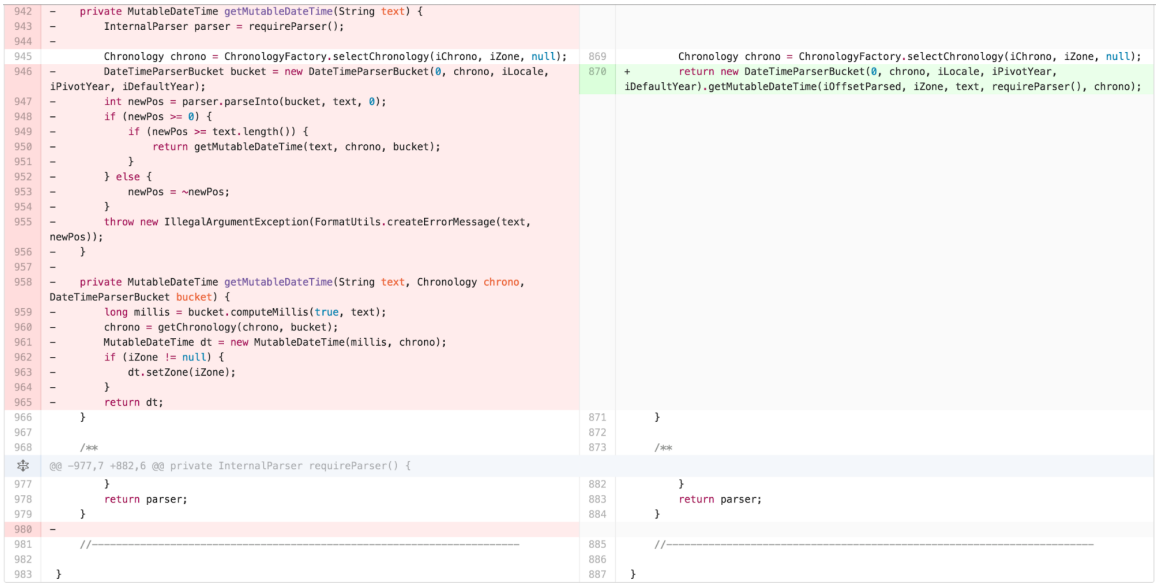


Figure C.49: insert caption

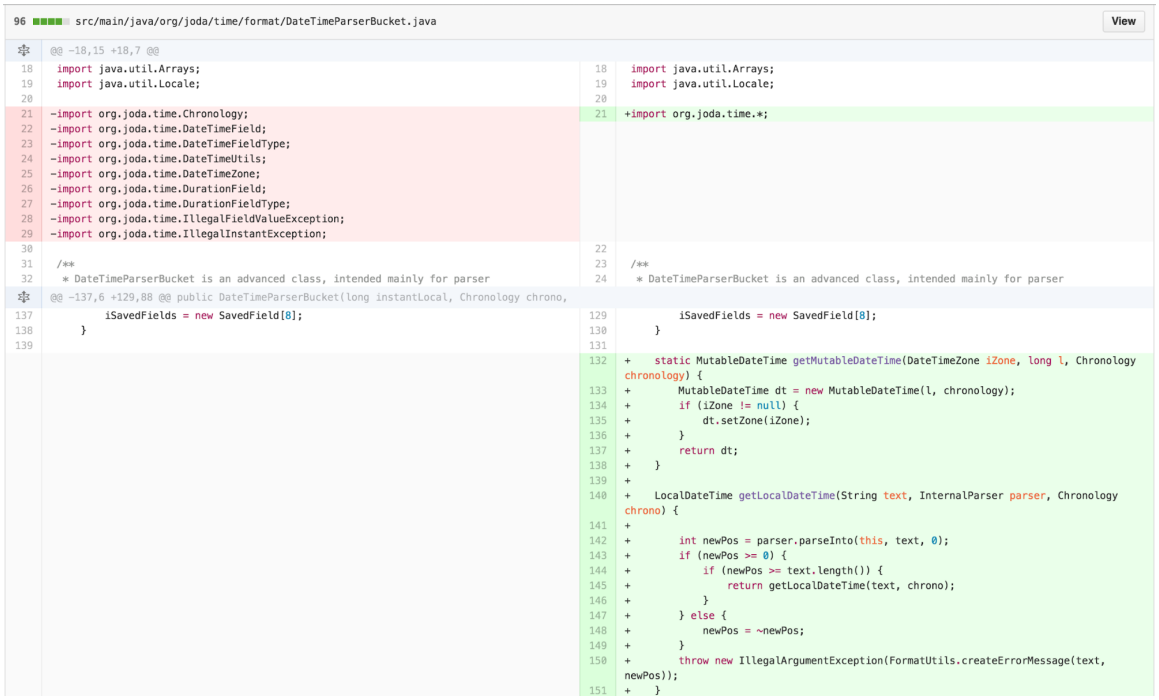
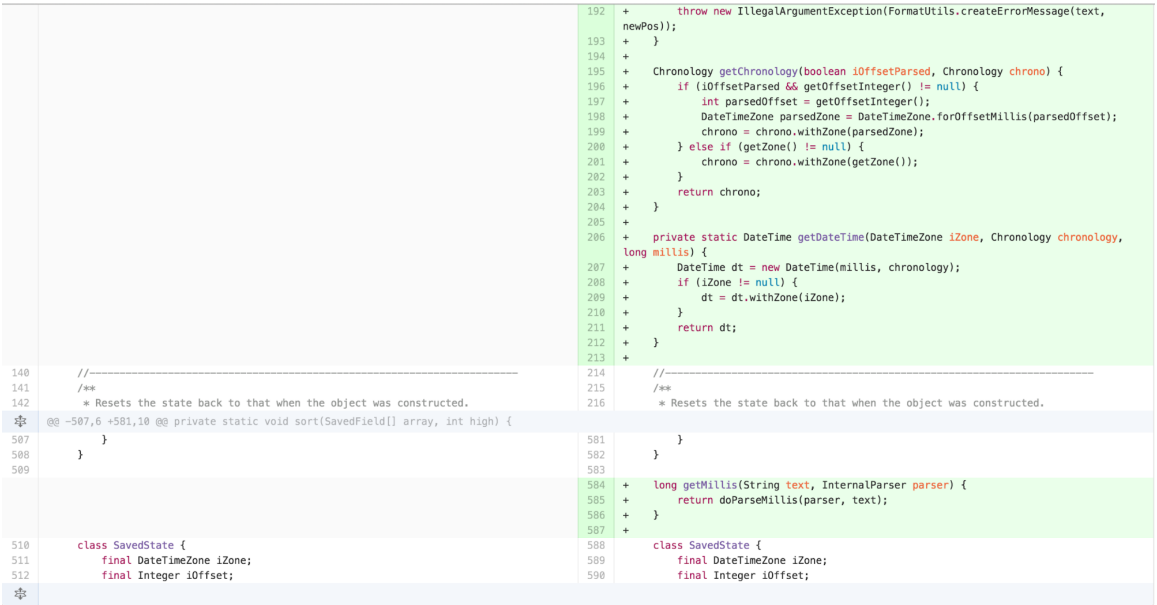


Figure C.50: insert caption

```
+
+    LocalDateTime getLocalDateTime(String text, Chronology chrono) {
+        long millis = computeMillis(true, text);
+        if (getOffsetInteger() != null) {  // treat withOffsetParsed() as being true
+            int parsedOffset = getOffsetInteger();
+            DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
+            chrono = chrono.withZone(parsedZone);
+        } else if (getZone() != null) {
+            chrono = chrono.withZone(getZone());
+        }
+        return new LocalDateTime(millis, chrono);
+    }
+
+    MutableDateTime getMutableDateTime(boolean iOffsetParsed, DateTimeZone iZone,
String text, InternalParser parser, Chronology chrono) {
+
+        int newPos = parser.parseInto(this, text, 0);
+        if (newPos >= 0) {
+            if (newPos >= text.length()) {
+                return getMutableDateTime(iOffsetParsed, iZone, text, chrono);
+            }
+        } else {
+            newPos = ~newPos;
+        }
+        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text,
newPos));
+    }
+
+    MutableDateTime getMutableDateTime(boolean iOffsetParsed, DateTimeZone iZone,
String text, Chronology chrono) {
+        return getMutableDateTime(iZone, computeMillis(true, text),
getChronology(iOffsetParsed, chrono));
+    }
+
+
+    DateTime getDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text,
final InternalParser parser, Chronology chrono) {
+        int newPos = parser.parseInto(this, text, 0);
+        if (newPos >= 0) {
+            if (newPos >= text.length()) {
+                return getDateTime(iZone, getChronology(iOffsetParsed, chrono),
computeMillis(true, text));
+            }
+        } else {
+            newPos = ~newPos;
+        }
```

Figure C.51: insert caption



```
+        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text,
newPos));
+    }
+
+    Chronology getChronology(boolean iOffsetParsed, Chronology chrono) {
+        if (iOffsetParsed && getOffsetInteger() != null) {
+            int parsedOffset = getOffsetInteger();
+            DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
+            chrono = chrono.withZone(parsedZone);
+        } else if (getZone() != null) {
+            chrono = chrono.withZone(getZone());
+        }
+        return chrono;
+    }
+
+    private static DateTime getDateTime(DateTimeZone iZone, Chronology chronology,
long millis) {
+        DateTime dt = new DateTime(millis, chronology);
+        if (iZone != null) {
+            dt = dt.withZone(iZone);
+        }
+        return dt;
+    }
+
     //-----------------------------------------------------------------------
     /**
      * Resets the state back to that when the object was constructed.
@@ -507,6 +581,10 @@ private static void sort(SavedField[] array, int high) {
         }
     }

+    long getMillis(String text, InternalParser parser) {
+        return doParseMillis(parser, text);
+    }
+
     class SavedState {
         final DateTimeZone iZone;
         final Integer iOffset;
```

Figure C.52: insert caption

At this point, I recognized that much of the logic in `parseIntoInstant` in `DateTimeFormatter` was similar to the other parsing methods. `DateTimeFormatter` also found itself doing

more work in its internals than necessary because the `ReadWriteableInstant` interface was missing an update feature. As that functionality seems implied by the name, I added it to the interface and pushed some responsibility onto `MutableDateTime`.

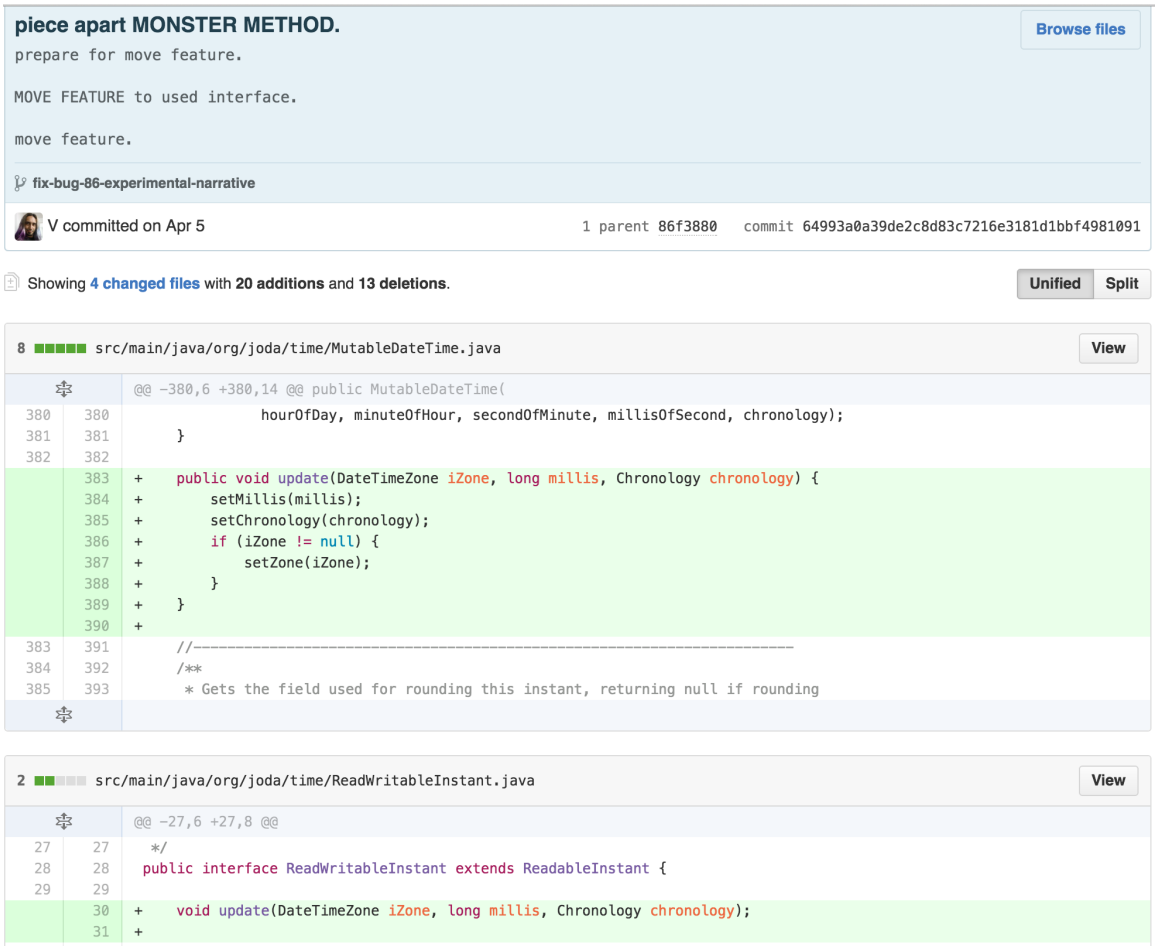


Figure C.53: insert caption

```
30  32      /**
31  33       * Sets the value as the number of milliseconds since
32  34       * the epoch, 1970-01-01T00:00:00Z.
```

17 src/main/java/org/joda/time/format/DateTimeFormatter.java View

```
@@ -712,47 +712,38 @@
712 712       * fields before calling this method, or use {@link #parseDateTime(String)}
713 713       * or {@link #parseMutableDateTime(String)}.
714 714       * <p>
715 715       * If it fails, the return value is negative, but the instant may still be
716 716       * modified. To determine the position where the parse failed, apply the
717 717       * one's complement operator (~) on the return value.
718 718       * <p>
719 719       * This parse method ignores the {@link #getDefaultYear() default year} and
720 720       * parses using the year from the supplied instant based on the chronology
721 721       * and time-zone of the supplied instant.
722 722       * <p>
723 723       * The parse will use the chronology of the instant.
724 724       *
725 725       * @param instant  an instant that will be modified, not null
726 726       * @param text  the text to parse
727 727       * @param position  position to start parsing from
728 728       * @return new position, negative value means parse failed -
729 729       *  apply complement operator (~) to get position of failure
730 730       * @throws UnsupportedOperationException if parsing is not supported
731 731       * @throws IllegalArgumentException if the instant is null
732 732       * @throws IllegalArgumentException if any field is out of range
733 733       */
734 734      public int parseInto(ReadWritableInstant instant, String text, int position) {
735     -        InternalParser parser = requireParser();
736 735          if (instant == null) {
737 736              throw new IllegalArgumentException("Instant must not be null");
738 737          }
739     -
740     -        long instantMillis = instant.getMillis();
741 738          Chronology chrono = instant.getChronology();
    739 +        long instantMillis = instant.getMillis();
742 740          int defaultYear = DateTimeUtils.getChronology(chrono).year().get(instantMillis);
743 741          long instantLocal = instantMillis + chrono.getZone().getOffset(instantMillis);
744 742          chrono = ChronologyFactory.selectChronology(iChrono, iZone, chrono);
745     -
    743 +
```

Figure C.54: insert caption

```
746 744          DateTimeParserBucket bucket = new DateTimeParserBucket(
747     -            instantLocal, chrono, iLocale, iPivotYear, defaultYear);
748     -        int newPos = parser.parseInto(bucket, text, position);
749     -        instant.setMillis(bucket.computeMillis(false, text));
750     -        chrono = bucket.getChronology(iOffsetParsed, chrono);
751     -        instant.setChronology(chrono);
752     -        if (iZone != null) {
753     -            instant.setZone(iZone);
754     -        }
755     -        return newPos;
    745 +                instantLocal, chrono, iLocale, iPivotYear, defaultYear);
    746 +        return bucket.parseIntoInstant(iOffsetParsed, iZone, instant, text, position, requireParser(), chrono);
756 747      }
757 748
758 749      /**
```

6 ■■■■■ src/main/java/org/joda/time/format/DateTimeParserBucket.java **View**

```
@@ -137,6 +137,12 @@ static MutableDateTime getMutableDateTime(DateTimeZone iZone, long l, Chronology
137 137          return dt;
138 138      }
139 139
    140 +    int parseIntoInstant(boolean iOffsetParsed, DateTimeZone iZone, ReadWritableInstant instant, String text, int posit
    141 +        int newPos = parser.parseInto(this, text, position);
    142 +        instant.update(iZone, computeMillis(false, text), getChronology(iOffsetParsed, chrono));
    143 +        return newPos;
    144 +    }
    145 +
140 146      LocalDateTime getLocalDateTime(String text, InternalParser parser, Chronology chrono) {
141 147
142 148          int newPos = parser.parseInto(this, text, 0);
```

Figure C.55: insert caption

At this point, I was able to push the entire responsibility of parsing into the `Instant` into `DateTimeParserBucket`.

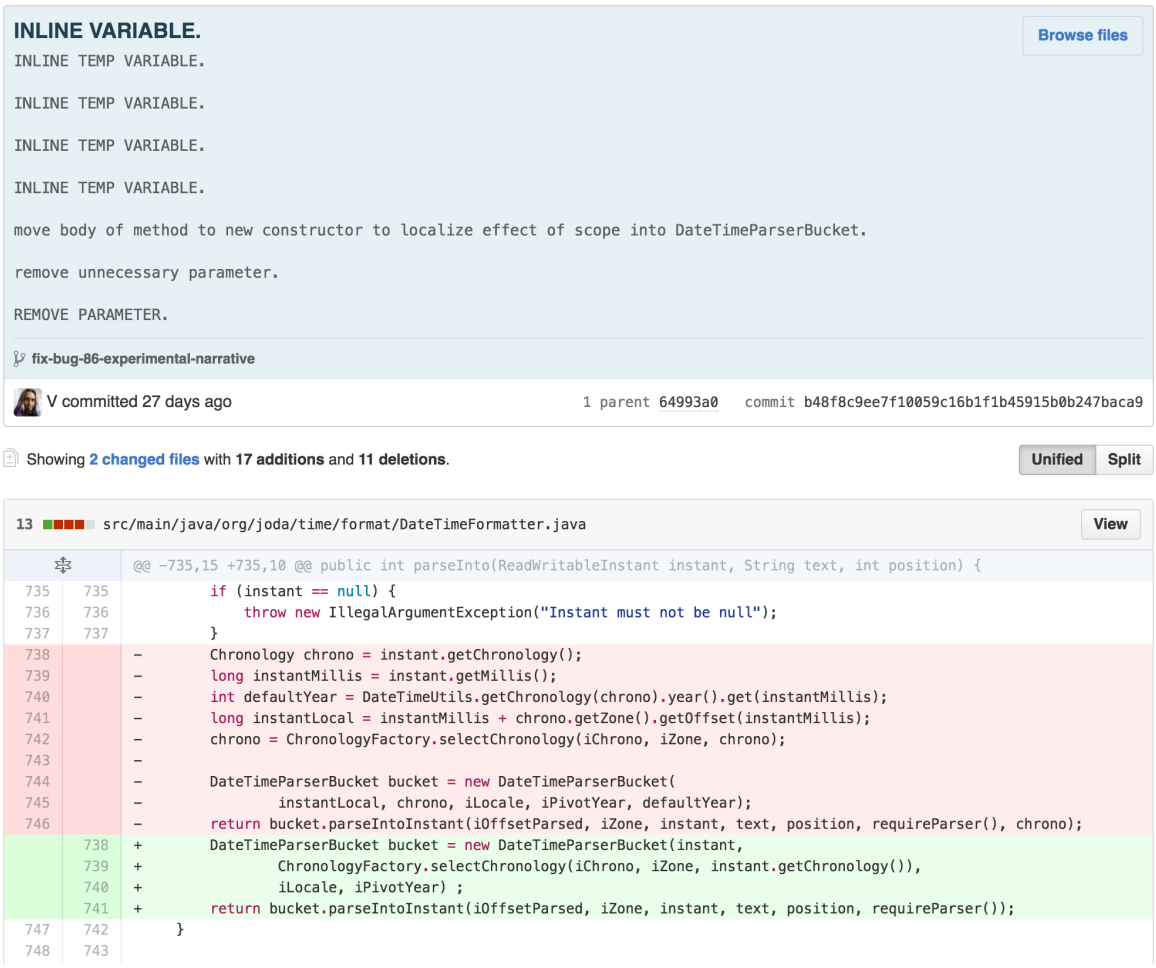


Figure C.56: insert caption

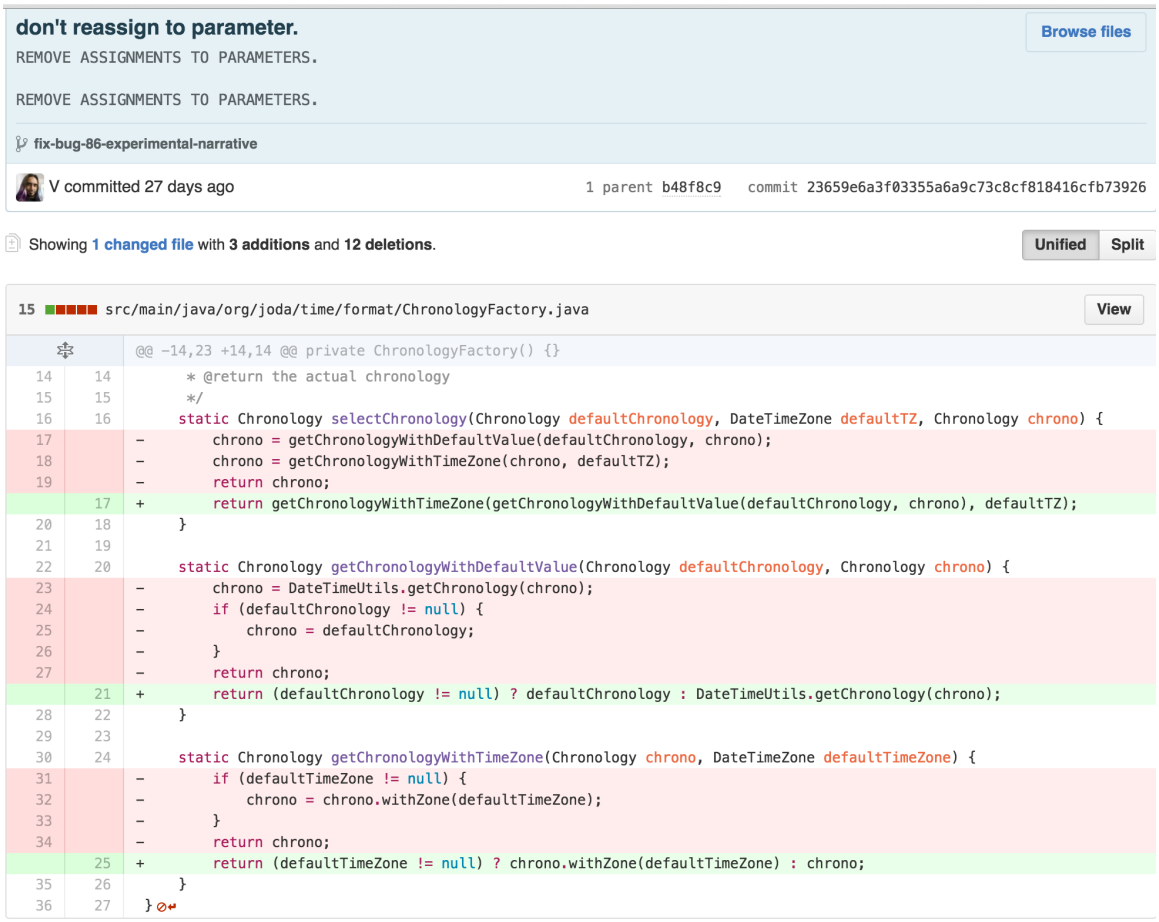
```
749  744      /**
```

15 src/main/java/org/joda/time/format/DateTimeParserBucket.java View

```
@@ -102,6 +102,17 @@ public DateTimeParserBucket(long instantLocal, Chronology chrono, Locale locale,
102  102          this(instantLocal, chrono, locale, pivotYear, 2000);
103  103      }
104  104
     105  +    public DateTimeParserBucket(ReadWritableInstant instant,
     106  +                                Chronology iChrono,
     107  +                                Locale iLocale,
     108  +                                Integer iPivotYear) {
     109  +        this(instant.getMillis() + instant.getChronology().getZone().getOffset(instant.getMillis()),
     110  +             iChrono,
     111  +             iLocale,
     112  +             iPivotYear,
     113  +             DateTimeUtils.getChronology(instant.getChronology()).year().get(instant.getMillis())
     114  +        );
     115  +    }
105  116      /**
106  117       * Constructs a bucket, with the option of specifying the pivot year for
107  118       * two-digit year parsing.
@@ -137,9 +148,9 @@ static MutableDateTime getMutableDateTime(DateTimeZone iZone, long l, Chronology
137  148          return dt;
138  149      }
139  150
140      -    int parseIntoInstant(boolean iOffsetParsed, DateTimeZone iZone, ReadWritableInstant instant, String text, int posit
     151  +    int parseIntoInstant(boolean iOffsetParsed, DateTimeZone iZone, ReadWritableInstant instant, String text, int posit
141  152          int newPos = parser.parseInto(this, text, position);
142      -        instant.update(iZone, computeMillis(false, text), getChronology(iOffsetParsed, chrono));
     153  +        instant.update(iZone, computeMillis(false, text), getChronology(iOffsetParsed, iChrono));
143  154          return newPos;
144  155      }
145  156
```

Figure C.57: insert caption

`DateTimeFormatter`'s parse methods were now doing very little besides using its state to set up a `DateTimeParserBucket`. It seemed like `DateTimeParserBucket` could be the permanent home of much of this functionality. I could move the functionality over. Along the way, I could start by removing some of the cognitive complexity of `ChronologyFactory`'s reassignment to input parameters simultaneously.



don't reassign to parameter.

Browse files

REMOVE ASSIGNMENTS TO PARAMETERS.

REMOVE ASSIGNMENTS TO PARAMETERS.

fix-bug-86-experimental-narrative

V committed 27 days ago — 1 parent b48f8c9 — commit 23659e6a3f03355a6a9c73c8cf818416cfb73926

Showing 1 changed file with 3 additions and 12 deletions. — Unified | Split

15 src/main/java/org/joda/time/format/ChronologyFactory.java — View

```
@@ -14,23 +14,14 @@ private ChronologyFactory() {}
14 14       * @return the actual chronology
15 15       */
16 16      static Chronology selectChronology(Chronology defaultChronology, DateTimeZone defaultTZ, Chronology chrono) {
17    -         chrono = getChronologyWithDefaultValue(defaultChronology, chrono);
18    -         chrono = getChronologyWithTimeZone(chrono, defaultTZ);
19    -         return chrono;
   17 +         return getChronologyWithTimeZone(getChronologyWithDefaultValue(defaultChronology, chrono), defaultTZ);
20 18      }
21 19
22 20      static Chronology getChronologyWithDefaultValue(Chronology defaultChronology, Chronology chrono) {
23    -         chrono = DateTimeUtils.getChronology(chrono);
24    -         if (defaultChronology != null) {
25    -             chrono = defaultChronology;
26    -         }
27    -         return chrono;
   21 +         return (defaultChronology != null) ? defaultChronology : DateTimeUtils.getChronology(chrono);
28 22      }
29 23
30 24      static Chronology getChronologyWithTimeZone(Chronology chrono, DateTimeZone defaultTimeZone) {
31    -         if (defaultTimeZone != null) {
32    -             chrono = chrono.withZone(defaultTimeZone);
33    -         }
34    -         return chrono;
   25 +         return (defaultTimeZone != null) ? chrono.withZone(defaultTimeZone) : chrono;
35 26      }
36 27  }⊘↵
```

Figure C.58: insert caption

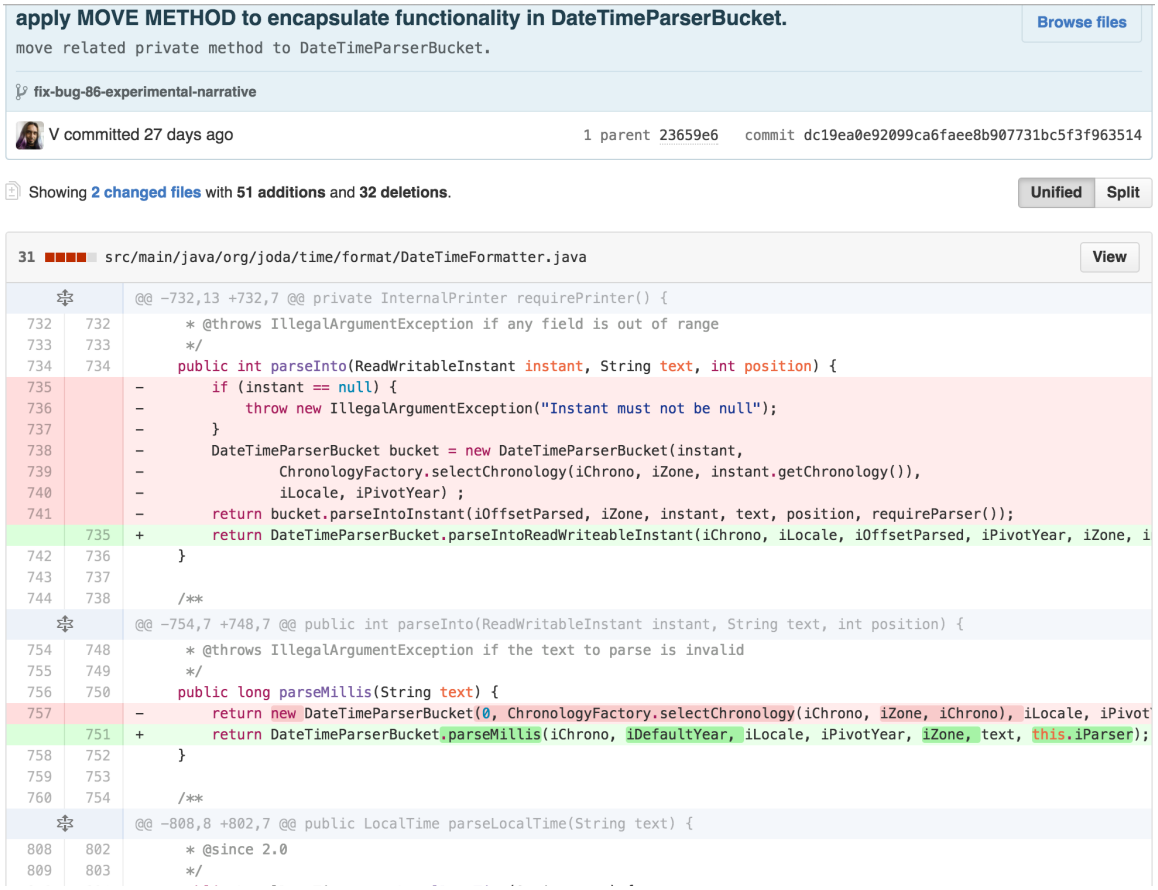
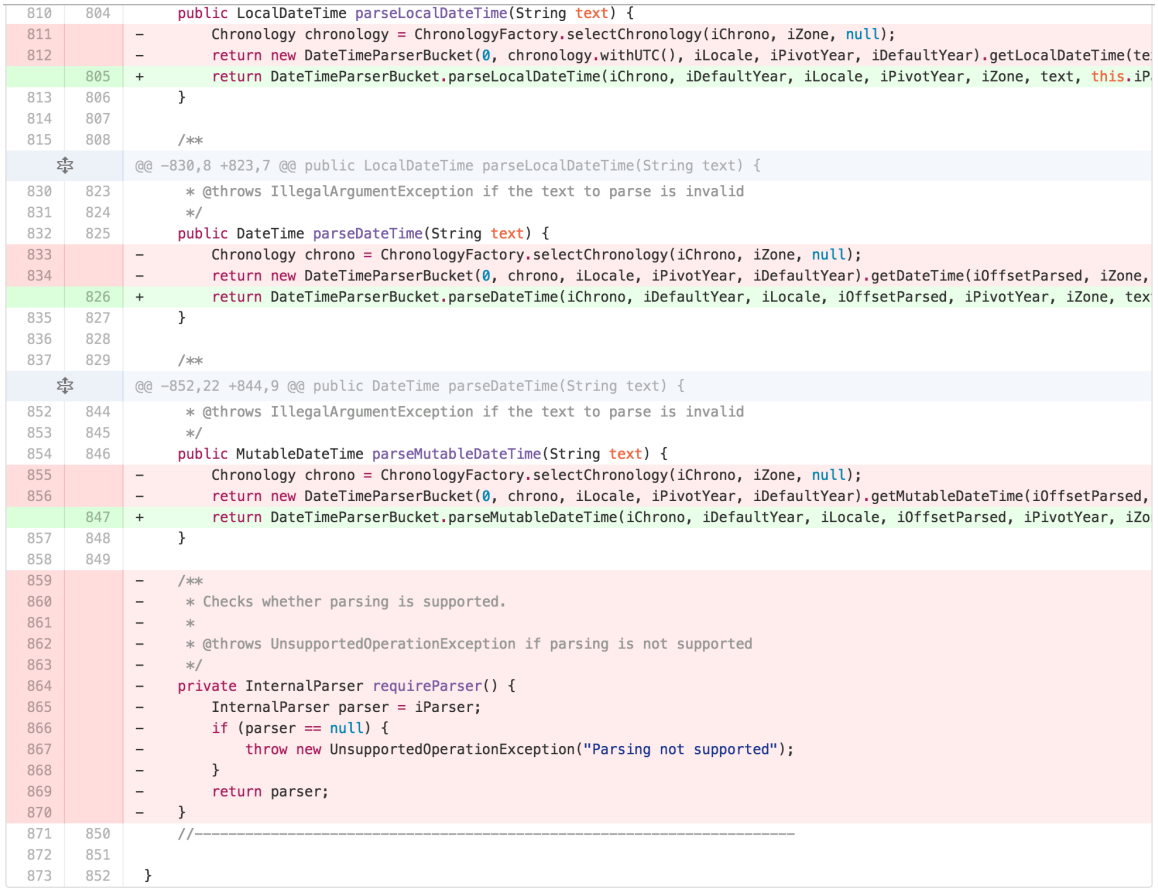


Figure C.59: insert caption



```
810  804      public LocalDateTime parseLocalDateTime(String text) {
811       -        Chronology chronology = ChronologyFactory.selectChronology(iChrono, iZone, null);
812       -        return new DateTimeParserBucket(0, chronology.withUTC(), iLocale, iPivotYear, iDefaultYear).getLocalDateTime(te
     805  +        return DateTimeParserBucket.parseLocalDateTime(iChrono, iDefaultYear, iLocale, iPivotYear, iZone, text, this.iP
813  806      }
814  807
815  808      /**
     @@ -830,8 +823,7 @@ public LocalDateTime parseLocalDateTime(String text) {
830  823       * @throws IllegalArgumentException if the text to parse is invalid
831  824       */
832  825      public DateTime parseDateTime(String text) {
833       -        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
834       -        return new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear).getDateTime(iOffsetParsed, iZone,
     826  +        return DateTimeParserBucket.parseDateTime(iChrono, iDefaultYear, iLocale, iOffsetParsed, iPivotYear, iZone, tex
835  827      }
836  828
837  829      /**
     @@ -852,22 +844,9 @@ public DateTime parseDateTime(String text) {
852  844       * @throws IllegalArgumentException if the text to parse is invalid
853  845       */
854  846      public MutableDateTime parseMutableDateTime(String text) {
855       -        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
856       -        return new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear).getMutableDateTime(iOffsetParsed,
     847  +        return DateTimeParserBucket.parseMutableDateTime(iChrono, iDefaultYear, iLocale, iOffsetParsed, iPivotYear, iZo
857  848      }
858  849
859       -    /**
860       -     * Checks whether parsing is supported.
861       -     *
862       -     * @throws UnsupportedOperationException if parsing is not supported
863       -     */
864       -    private InternalParser requireParser() {
865       -        InternalParser parser = iParser;
866       -        if (parser == null) {
867       -            throw new UnsupportedOperationException("Parsing not supported");
868       -        }
869       -        return parser;
870       -    }
871  850      //-----------------------------------------------------------------------
872  851
873  852  }
```

Figure C.60: insert caption

52 src/main/java/org/joda/time/format/DateTimeParserBucket.java View

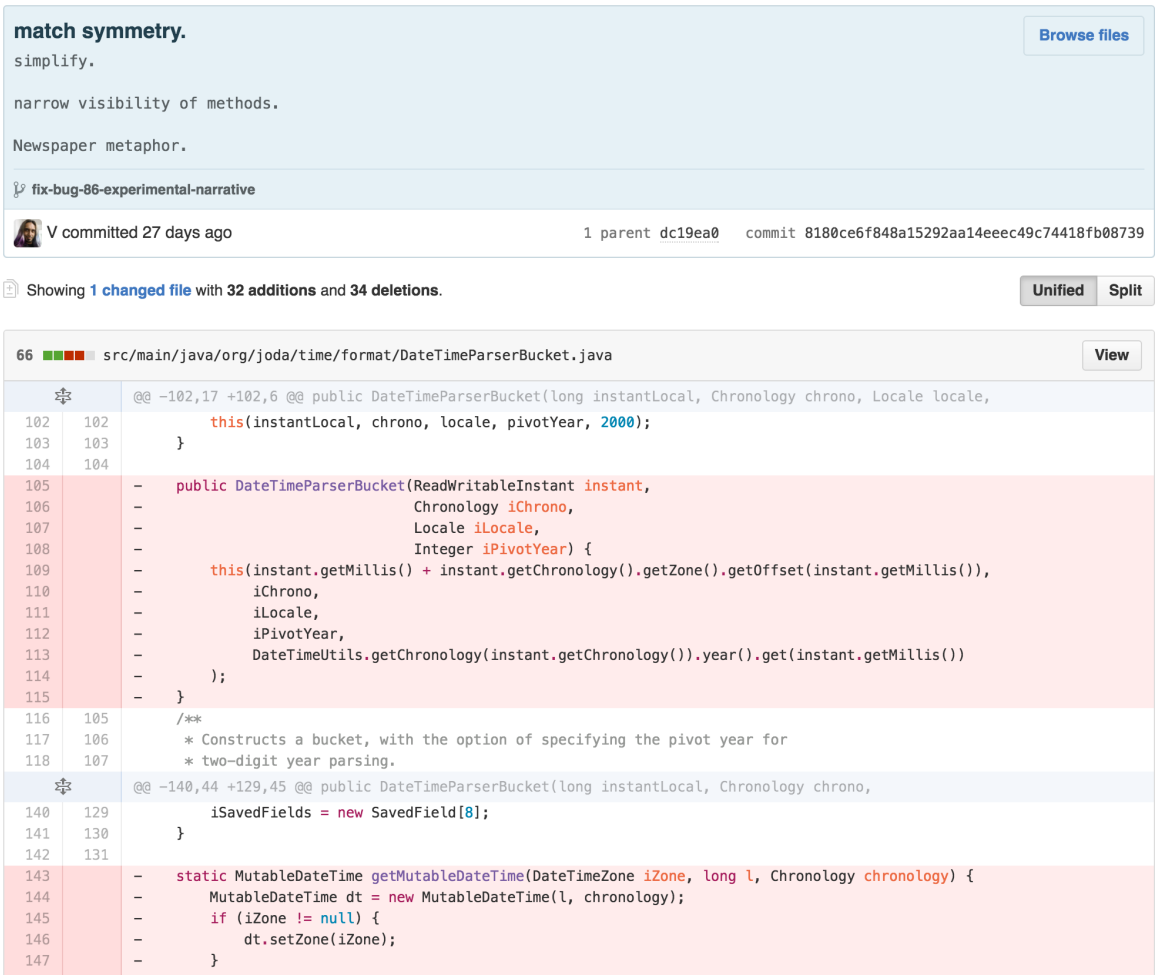
```
@@ -148,15 +148,55 @@ static MutableDateTime getMutableDateTime(DateTimeZone iZone, long l, Chronology
148 148         return dt;
149 149     }
150 150
151     -   int parseIntoInstant(boolean iOffsetParsed, DateTimeZone iZone, ReadWritableInstant instant, String text, int posit
152     -       int newPos = parser.parseInto(this, text, position);
153     -       instant.update(iZone, computeMillis(false, text), getChronology(iOffsetParsed, iChrono));
    151 +   static int parseIntoReadWriteableInstant(Chronology iChrono, Locale iLocale, boolean iOffsetParsed, Integer iPivotY
    152 +       if (instant == null) {
    153 +           throw new IllegalArgumentException("Instant must not be null");
    154 +       }
    155 +       DateTimeParserBucket bucket = new DateTimeParserBucket(instant,
    156 +               ChronologyFactory.selectChronology(iChrono, iZone, instant.getChronology()),
    157 +               iLocale, iPivotYear) ;
    158 +       int newPos = requireParser(parser).parseInto(bucket, text, position);
    159 +       instant.update(iZone, bucket.computeMillis(false, text), bucket.getChronology(iOffsetParsed, bucket.iChrono));
154 160         return newPos;
155 161     }
156 162
    163 +   static long parseMillis(Chronology iChrono, int iDefaultYear, Locale iLocale, Integer iPivotYear, DateTimeZone iZon
    164 +       Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, iChrono);
    165 +       return new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear).getMillis(text, parser);
    166 +   }
    167 +
    168 +   static LocalDateTime parseLocalDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, Integer iPivotYear, D
    169 +       Chronology chronology = ChronologyFactory.selectChronology(iChrono, iZone, null);
    170 +       return new DateTimeParserBucket(0, chronology.withUTC(), iLocale, iPivotYear, iDefaultYear).getLocalDateTime(te
    171 +   }
    172 +
    173 +   static DateTime parseDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, boolean iOffsetParsed, Integer
    174 +       Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
    175 +       return new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear).getDateTime(iOffsetParsed, iZone,
    176 +   }
    177 +
    178 +   static MutableDateTime parseMutableDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, boolean iOffsetPa
    179 +       Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
    180 +       return new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear).getMutableDateTime(iOffsetParsed,
    181 +   }
    182 +
```

Figure C.61: insert caption

```
+    /**
+     * Checks whether parsing is supported.
+     *
+     * @throws UnsupportedOperationException if parsing is not supported
+     * @param iParser
+     */
+    private static InternalParser requireParser(InternalParser iParser) {
+        InternalParser parser = iParser;
+        if (parser == null) {
+            throw new UnsupportedOperationException("Parsing not supported");
+        }
+        return parser;
+    }
+
     LocalDateTime getLocalDateTime(String text, InternalParser parser, Chronology chrono) {

-        int newPos = parser.parseInto(this, text, 0);
+        int newPos = requireParser(parser).parseInto(this, text, 0);
         if (newPos >= 0) {
             if (newPos >= text.length()) {
                 return getLocalDateTime(text, chrono);
@@ -181,7 +221,7 @@ LocalDateTime getLocalDateTime(String text, Chronology chrono) {

     MutableDateTime getMutableDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, InternalParser parser, C

-        int newPos = parser.parseInto(this, text, 0);
+        int newPos = requireParser(parser).parseInto(this, text, 0);
         if (newPos >= 0) {
             if (newPos >= text.length()) {
                 return getMutableDateTime(iOffsetParsed, iZone, text, chrono);
@@ -198,7 +238,7 @@ MutableDateTime getMutableDateTime(boolean iOffsetParsed, DateTimeZone iZone, St


     DateTime getDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, final InternalParser parser, Chronolog

-        int newPos = parser.parseInto(this, text, 0);
+        int newPos = requireParser(parser).parseInto(this, text, 0);
         if (newPos >= 0) {
             if (newPos >= text.length()) {
                 return getDateTime(iZone, getChronology(iOffsetParsed, chrono), computeMillis(true, text));
```

Figure C.62: insert caption

At this point, it felt like I was succeeding more in moving complexity around than removing it. `DateTimeParserBucket` had most of the functionality related to parsing moved to it, as its internal state was a big part of most of the calculations. But I was no closer to my originally envisioned TEMPLATE METHOD. I also worried that more responsibility on an originally used intermediate calculation scratchpad object may belie its importance in the process. I sought further simplification. I started by reducing the scope of some methods of `DateTimeParserBucket` and re-arranging them according to the Newspaper Metaphor.



match symmetry.

Browse files

simplify.

narrow visibility of methods.

Newspaper metaphor.

fix-bug-86-experimental-narrative

V committed 27 days ago 1 parent dc19ea0 commit 8180ce6f848a15292aa14eeec49c74418fb08739

Showing 1 changed file with 32 additions and 34 deletions. Unified Split

66 src/main/java/org/joda/time/format/DateTimeParserBucket.java View

```
@@ -102,17 +102,6 @@ public DateTimeParserBucket(long instantLocal, Chronology chrono, Locale locale,
102 102         this(instantLocal, chrono, locale, pivotYear, 2000);
103 103     }
104 104
105     -   public DateTimeParserBucket(ReadWritableInstant instant,
106     -                                Chronology iChrono,
107     -                                Locale iLocale,
108     -                                Integer iPivotYear) {
109     -       this(instant.getMillis() + instant.getChronology().getZone().getOffset(instant.getMillis()),
110     -            iChrono,
111     -            iLocale,
112     -            iPivotYear,
113     -            DateTimeUtils.getChronology(instant.getChronology()).year().get(instant.getMillis())
114     -       );
115     -   }
116 105     /**
117 106      * Constructs a bucket, with the option of specifying the pivot year for
118 107      * two-digit year parsing.
@@ -140,44 +129,45 @@ public DateTimeParserBucket(long instantLocal, Chronology chrono,
140 129         iSavedFields = new SavedField[8];
141 130     }
142 131
143     -   static MutableDateTime getMutableDateTime(DateTimeZone iZone, long l, Chronology chronology) {
144     -       MutableDateTime dt = new MutableDateTime(l, chronology);
145     -       if (iZone != null) {
146     -           dt.setZone(iZone);
147     -       }
```

Figure C.63: insert caption

```
148         -       return dt;
149         -   }
150         -
151   132       static int parseIntoReadWriteableInstant(Chronology iChrono, Locale iLocale, boolean iOffsetParsed, Integer iPivotYea
152   133           if (instant == null) {
153   134               throw new IllegalArgumentException("Instant must not be null");
154   135           }
155         -       DateTimeParserBucket bucket = new DateTimeParserBucket(instant,
156         -               ChronologyFactory.selectChronology(iChrono, iZone, instant.getChronology()),
157         -               iLocale, iPivotYear) ;
158         -       int newPos = requireParser(parser).parseInto(bucket, text, position);
159         -       instant.update(iZone, bucket.computeMillis(false, text), bucket.getChronology(iOffsetParsed, bucket.iChrono));
160   136 +       Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, instant.getChronology());
161   137 +       long millis = instant.getMillis() + instant.getChronology().getZone().getOffset(instant.getMillis());
162   138 +       int defaultYear = DateTimeUtils.getChronology(instant.getChronology()).year().get(instant.getMillis());
163   139 +       DateTimeParserBucket bucket = new DateTimeParserBucket(millis, chrono, iLocale, iPivotYear, defaultYear);
164   140 +       return bucket.parseIntoInstantAndGetPosition(iOffsetParsed, iZone, instant, text, position, parser);
165   141 +   }
166   142 +
167   143 +   private int parseIntoInstantAndGetPosition(boolean iOffsetParsed, DateTimeZone iZone, ReadWritableInstant instant, St
168   144 +       int newPos = requireParser(parser).parseInto(this, text, position);
169   145 +       instant.update(iZone, computeMillis(false, text), getChronology(iOffsetParsed, iChrono));
170   146           return newPos;
171   147       }
172   148
173   149       static long parseMillis(Chronology iChrono, int iDefaultYear, Locale iLocale, Integer iPivotYear, DateTimeZone iZone,
174   150           Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, iChrono);
175         -       return new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear).getMillis(text, parser);
176   151 +       DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
177   152 +       return bucket.getMillis(text, parser);
178   153       }
179   154
180   155       static LocalDateTime parseLocalDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, Integer iPivotYear, Dat
181   156           Chronology chronology = ChronologyFactory.selectChronology(iChrono, iZone, null);
182         -       return new DateTimeParserBucket(0, chronology.withUTC(), iLocale, iPivotYear, iDefaultYear).getLocalDateTime(text
183   157 +       DateTimeParserBucket bucket = new DateTimeParserBucket(0, chronology.withUTC(), iLocale, iPivotYear, iDefaultYea
184   158 +       return bucket.getLocalDateTime(text, parser, chronology.withUTC());
185   159       }
186   160
187   161       static DateTime parseDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, boolean iOffsetParsed, Integer iP
188   162           Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
189         -       return new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear).getDateTime(iOffsetParsed, iZone, t
190   163 +       DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
191   164 +       return bucket.getDateTime(iOffsetParsed, iZone, text, parser, chrono);
192   165       }
193   166
```

Figure C.64: insert caption

Figure C.65: insert caption

```
    229 +    static MutableDateTime getMutableDateTime(DateTimeZone iZone, long l, Chronology chronology) {
    230 +        MutableDateTime dt = new MutableDateTime(l, chronology);
    231 +        if (iZone != null) {
    232 +            dt.setZone(iZone);
    233 +        }
    234 +        return dt;
    235 +    }
    236 +
239 237
240     -    DateTime getDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, final InternalParser parser, Chronology
        238 +    private DateTime getDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, final InternalParser parser, Ch
241 239            int newPos = requireParser(parser).parseInto(this, text, 0);
242 240            if (newPos >= 0) {
243 241                if (newPos >= text.length()) {
    @@ -249,7 +247,7 @@ DateTime getDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, fin
249 247            throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
250 248        }
251 249
252     -    Chronology getChronology(boolean iOffsetParsed, Chronology chrono) {
        250 +    private Chronology getChronology(boolean iOffsetParsed, Chronology chrono) {
253 251            if (iOffsetParsed && getOffsetInteger() != null) {
254 252                int parsedOffset = getOffsetInteger();
255 253                DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
```

Figure C.66: insert caption

This helped me identify that the best approach for getting to the desired architecture was to split apart the functionality that had migrated into `DateTimeParserBucket` into another collaborating object.

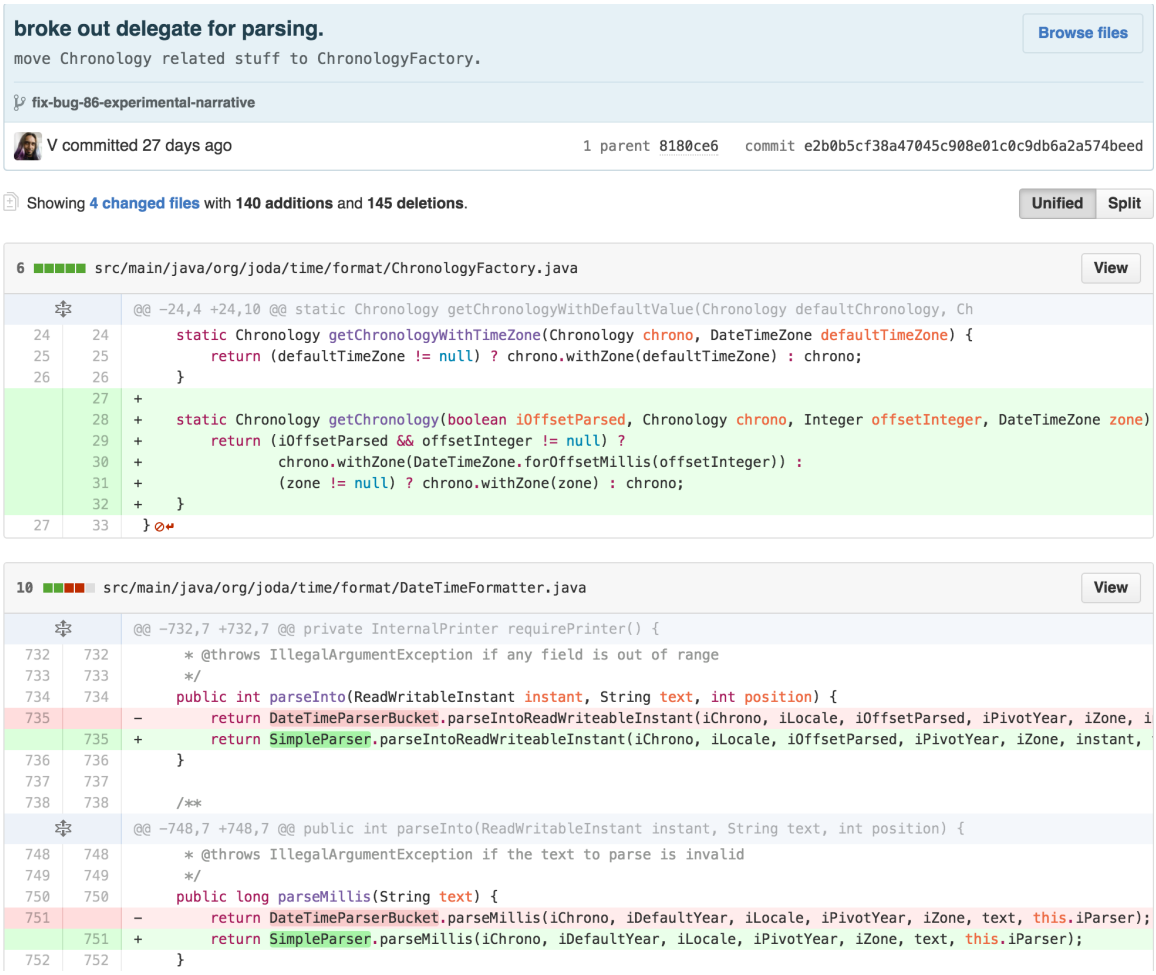


Figure C.67: insert caption

Figure C.68: insert caption

140 src/main/java/org/joda/time/format/DateTimeParserBucket.java View

```
@@ -129,142 +129,6 @@ public DateTimeParserBucket(long instantLocal, Chronology chrono,
         iSavedFields = new SavedField[8];
     }

-    static int parseIntoReadWriteableInstant(Chronology iChrono, Locale iLocale, boolean iOffsetParsed, Integer iPivotY
-        if (instant == null) {
-            throw new IllegalArgumentException("Instant must not be null");
-        }
-        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, instant.getChronology());
-        long millis = instant.getMillis() + instant.getChronology().getZone().getOffset(instant.getMillis());
-        int defaultYear = DateTimeUtils.getChronology(instant.getChronology()).year().get(instant.getMillis());
-        DateTimeParserBucket bucket = new DateTimeParserBucket(millis, chrono, iLocale, iPivotYear, defaultYear);
-        return bucket.parseIntoInstantAndGetPosition(iOffsetParsed, iZone, instant, text, position, parser);
-    }
-
-    private int parseIntoInstantAndGetPosition(boolean iOffsetParsed, DateTimeZone iZone, ReadWritableInstant instant,
-        int newPos = requireParser(parser).parseInto(this, text, position);
-        instant.update(iZone, computeMillis(false, text), getChronology(iOffsetParsed, iChrono));
-        return newPos;
-    }
-
-    static long parseMillis(Chronology iChrono, int iDefaultYear, Locale iLocale, Integer iPivotYear, DateTimeZone iZon
-        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, iChrono);
-        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
-        return bucket.getMillis(text, parser);
-    }
-
-    static LocalDateTime parseLocalDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, Integer iPivotYear, D
-        Chronology chronology = ChronologyFactory.selectChronology(iChrono, iZone, null);
-        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chronology.withUTC(), iLocale, iPivotYear, iDefaultYe
-        return bucket.getLocalDateTime(text, parser, chronology.withUTC());
-    }
-
-    static DateTime parseDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, boolean iOffsetParsed, Integer
-        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
-        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
-        return bucket.getDateTime(iOffsetParsed, iZone, text, parser, chrono);
-    }
-
-    static MutableDateTime parseMutableDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, boolean iOffsetPa
-        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
-        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
-        return bucket.getMutableDateTime(iOffsetParsed, iZone, text, parser, chrono);
```

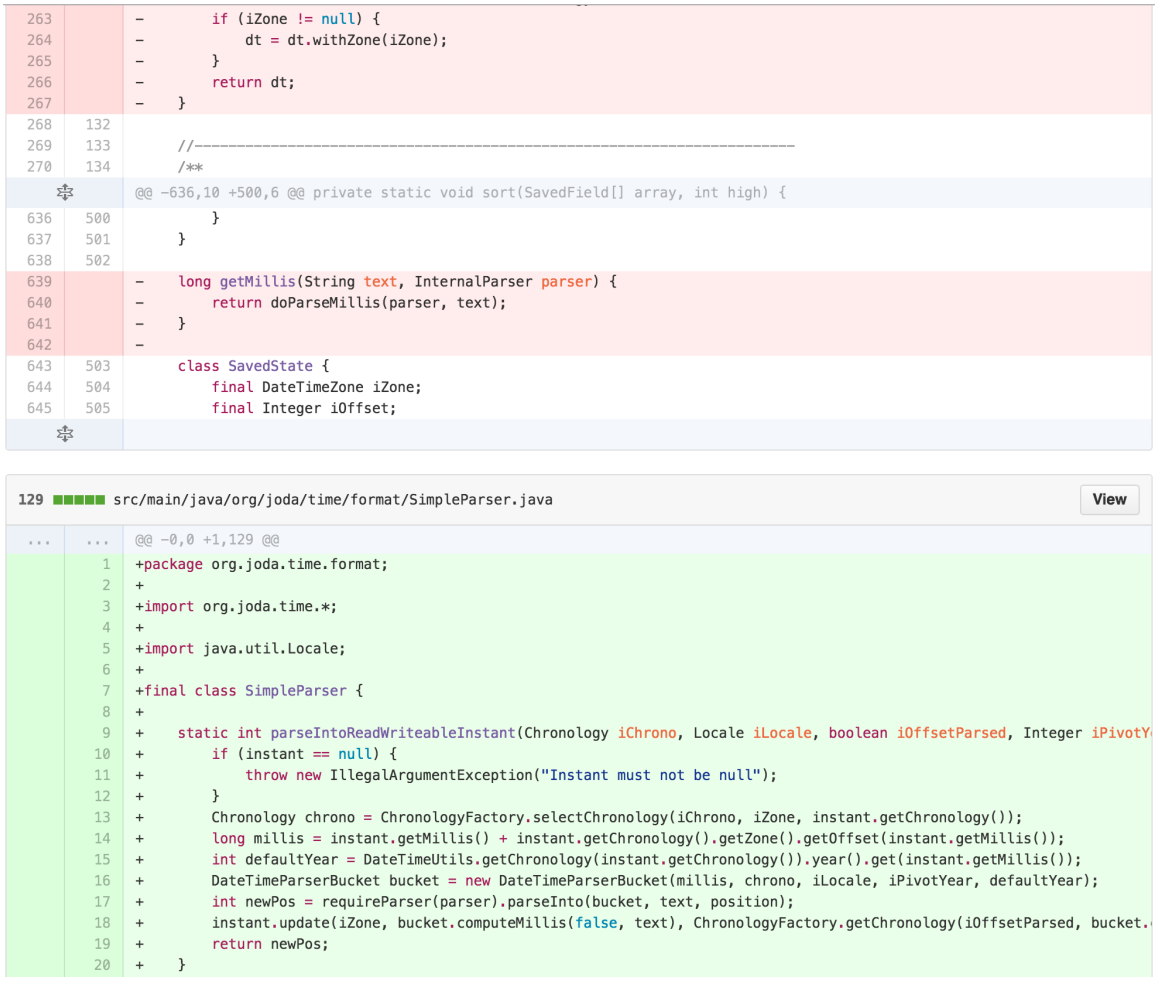
Figure C.69: insert caption

```
171 -    }
172 -
173 -    /**
174 -     * Checks whether parsing is supported.
175 -     *
176 -     * @throws UnsupportedOperationException if parsing is not supported
177 -     * @param iParser
178 -     */
179 -    private static InternalParser requireParser(InternalParser iParser) {
180 -        InternalParser parser = iParser;
181 -        if (parser == null) {
182 -            throw new UnsupportedOperationException("Parsing not supported");
183 -        }
184 -        return parser;
185 -    }
186 -
187 -    private LocalDateTime getLocalDateTime(String text, InternalParser parser, Chronology chrono) {
188 -
189 -        int newPos = requireParser(parser).parseInto(this, text, 0);
190 -        if (newPos >= 0) {
191 -            if (newPos >= text.length()) {
192 -                return getLocalDateTime(text, chrono);
193 -            }
194 -        } else {
195 -            newPos = ~newPos;
196 -        }
197 -        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
198 -    }
199 -
200 -    private LocalDateTime getLocalDateTime(String text, Chronology chrono) {
201 -        long millis = computeMillis(true, text);
202 -        if (getOffsetInteger() != null) {  // treat withOffsetParsed() as being true
203 -            int parsedOffset = getOffsetInteger();
204 -            DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
205 -            chrono = chrono.withZone(parsedZone);
206 -        } else if (getZone() != null) {
207 -            chrono = chrono.withZone(getZone());
208 -        }
209 -        return new LocalDateTime(millis, chrono);
210 -    }
211 -
212 -    private MutableDateTime getMutableDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, InternalParser p
213 -
214 -        int newPos = requireParser(parser).parseInto(this, text, 0);
215 -        if (newPos >= 0) {
216 -            if (newPos >= text.length()) {
```

Figure C.70: insert caption

```
217         -                return getMutableDateTime(iOffsetParsed, iZone, text, chrono);
218         -            }
219         -        } else {
220         -            newPos = ~newPos;
221         -        }
222         -        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
223         -    }
224         -
225         -    private MutableDateTime getMutableDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, Chronology chron
226         -        return getMutableDateTime(iZone, computeMillis(true, text), getChronology(iOffsetParsed, chrono));
227         -    }
228         -
229         -    static MutableDateTime getMutableDateTime(DateTimeZone iZone, long l, Chronology chronology) {
230         -        MutableDateTime dt = new MutableDateTime(l, chronology);
231         -        if (iZone != null) {
232         -            dt.setZone(iZone);
233         -        }
234         -        return dt;
235         -    }
236         -
237         -
238         -    private DateTime getDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, final InternalParser parser, C
239         -        int newPos = requireParser(parser).parseInto(this, text, 0);
240         -        if (newPos >= 0) {
241         -            if (newPos >= text.length()) {
242         -                return getDateTime(iZone, getChronology(iOffsetParsed, chrono), computeMillis(true, text));
243         -            }
244         -        } else {
245         -            newPos = ~newPos;
246         -        }
247         -        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
248         -    }
249         -
250         -    private Chronology getChronology(boolean iOffsetParsed, Chronology chrono) {
251         -        if (iOffsetParsed && getOffsetInteger() != null) {
252         -            int parsedOffset = getOffsetInteger();
253         -            DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
254         -            chrono = chrono.withZone(parsedZone);
255         -        } else if (getZone() != null) {
256         -            chrono = chrono.withZone(getZone());
257         -        }
258         -        return chrono;
259         -    }
260         -
261         -    private static DateTime getDateTime(DateTimeZone iZone, Chronology chronology, long millis) {
262         -        DateTime dt = new DateTime(millis, chronology);
```

Figure C.71: insert caption



```
263        -        if (iZone != null) {
264        -            dt = dt.withZone(iZone);
265        -        }
266        -        return dt;
267        -    }
268  132
269  133       //-----------------------------------------------------------------------
270  134       /**
          @@ -636,10 +500,6 @@ private static void sort(SavedField[] array, int high) {
636  500           }
637  501       }
638  502
639        -    long getMillis(String text, InternalParser parser) {
640        -        return doParseMillis(parser, text);
641        -    }
642        -
643  503       class SavedState {
644  504           final DateTimeZone iZone;
645  505           final Integer iOffset;
```

129 ■■■■■ src/main/java/org/joda/time/format/SimpleParser.java View

```
...  ...   @@ -0,0 +1,129 @@
   1  +package org.joda.time.format;
   2  +
   3  +import org.joda.time.*;
   4  +
   5  +import java.util.Locale;
   6  +
   7  +final class SimpleParser {
   8  +
   9  +    static int parseIntoReadWriteableInstant(Chronology iChrono, Locale iLocale, boolean iOffsetParsed, Integer iPivotY
  10  +        if (instant == null) {
  11  +            throw new IllegalArgumentException("Instant must not be null");
  12  +        }
  13  +        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, instant.getChronology());
  14  +        long millis = instant.getMillis() + instant.getChronology().getZone().getOffset(instant.getMillis());
  15  +        int defaultYear = DateTimeUtils.getChronology(instant.getChronology()).year().get(instant.getMillis());
  16  +        DateTimeParserBucket bucket = new DateTimeParserBucket(millis, chrono, iLocale, iPivotYear, defaultYear);
  17  +        int newPos = requireParser(parser).parseInto(bucket, text, position);
  18  +        instant.update(iZone, bucket.computeMillis(false, text), ChronologyFactory.getChronology(iOffsetParsed, bucket.
  19  +        return newPos;
  20  +    }
```

Figure C.72: insert caption

```
+
+    static long parseMillis(Chronology iChrono, int iDefaultYear, Locale iLocale, Integer iPivotYear, DateTimeZone iZon
+        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, iChrono);
+        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
+        return bucket.doParseMillis(parser, text);
+    }
+
+    static LocalDateTime parseLocalDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, Integer iPivotYear, D
+        Chronology chronology = ChronologyFactory.selectChronology(iChrono, iZone, null);
+        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chronology.withUTC(), iLocale, iPivotYear, iDefaultYe
+        return getLocalDateTime(bucket, text, parser, chronology.withUTC());
+    }
+
+    static DateTime parseDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, boolean iOffsetParsed, Integer
+        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
+        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
+        return getDateTime(bucket, iOffsetParsed, iZone, text, parser, chrono);
+    }
+
+    static MutableDateTime parseMutableDateTime(Chronology iChrono, int iDefaultYear, Locale iLocale, boolean iOffsetPa
+        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, null);
+        DateTimeParserBucket bucket = new DateTimeParserBucket(0, chrono, iLocale, iPivotYear, iDefaultYear);
+        return getMutableDateTime(bucket, iOffsetParsed, iZone, text, parser, chrono);
+    }
+
+    /**
+     * Checks whether parsing is supported.
+     *
+     * @param iParser
+     * @throws UnsupportedOperationException if parsing is not supported
+     */
+    static InternalParser requireParser(InternalParser iParser) {
+        InternalParser parser = iParser;
+        if (parser == null) {
+            throw new UnsupportedOperationException("Parsing not supported");
+        }
+        return parser;
+    }
+
+    static LocalDateTime getLocalDateTime(DateTimeParserBucket bucket, String text, InternalParser parser, Chronology c
+
+        int newPos = requireParser(parser).parseInto(bucket, text, 0);
+        if (newPos >= 0) {
+            if (newPos >= text.length()) {
+                return getLocalDateTime(bucket, text, chrono);
+            }
```

Figure C.73: insert caption

```
+        } else {
+            newPos = ~newPos;
+        }
+        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
+    }
+
+    static LocalDateTime getLocalDateTime(DateTimeParserBucket bucket, String text, Chronology chrono) {
+        long millis = bucket.computeMillis(true, text);
+        if (bucket.getOffsetInteger() != null) {  // treat withOffsetParsed() as being true
+            int parsedOffset = bucket.getOffsetInteger();
+            DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
+            chrono = chrono.withZone(parsedZone);
+        } else if (bucket.getZone() != null) {
+            chrono = chrono.withZone(bucket.getZone());
+        }
+        return new LocalDateTime(millis, chrono);
+    }
+
+    static MutableDateTime getMutableDateTime(DateTimeParserBucket bucket, boolean iOffsetParsed, DateTimeZone iZone, S
+
+        int newPos = requireParser(parser).parseInto(bucket, text, 0);
+        if (newPos >= 0) {
+            if (newPos >= text.length()) {
+                return getMutableDateTime(bucket, iOffsetParsed, iZone, text, chrono);
+            }
+        } else {
+            newPos = ~newPos;
+        }
+        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
+    }
+
+    static MutableDateTime getMutableDateTime(DateTimeParserBucket bucket, boolean iOffsetParsed, DateTimeZone iZone, S
+        return getMutableDateTime(iZone, bucket.computeMillis(true, text), ChronologyFactory.getChronology(iOffsetParse
+    }
+
+    static MutableDateTime getMutableDateTime(DateTimeZone iZone, long l, Chronology chronology) {
+        MutableDateTime dt = new MutableDateTime(l, chronology);
+        if (iZone != null) {
+            dt.setZone(iZone);
+        }
+        return dt;
+    }
+
+    static DateTime getDateTime(DateTimeParserBucket bucket, boolean iOffsetParsed, DateTimeZone iZone, String text, fi
+        int newPos = requireParser(parser).parseInto(bucket, text, 0);
+        if (newPos >= 0) {
```

Figure C.74: insert caption

```
+            if (newPos >= text.length()) {
+                return getDateTime(iZone, ChronologyFactory.getChronology(iOffsetParsed, chrono, bucket.getOffsetInteger(), buck
+            }
+        } else {
+            newPos = ~newPos;
+        }
+        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
+    }
+
+    static DateTime getDateTime(DateTimeZone iZone, Chronology chronology, long millis) {
+        DateTime dt = new DateTime(millis, chronology);
+        if (iZone != null) {
+            dt = dt.withZone(iZone);
+        }
+        return dt;
+    }
+}
```

Figure C.75: insert caption

With the new delegate `SimpleParser` in place, I had a target to move towards the desired TEMPLATE METHOD hierarchy. First, I started by inlining some meth-

ods I'd previously broken out. Refactoring operations are often well-known to be reversible, as exploring in one direction can lead to realization that the best path was another. I also applying some EXTRACT PARAMETER to reduce the length of method signatures, as the cognitive complexity of methods rises with the length its signature.

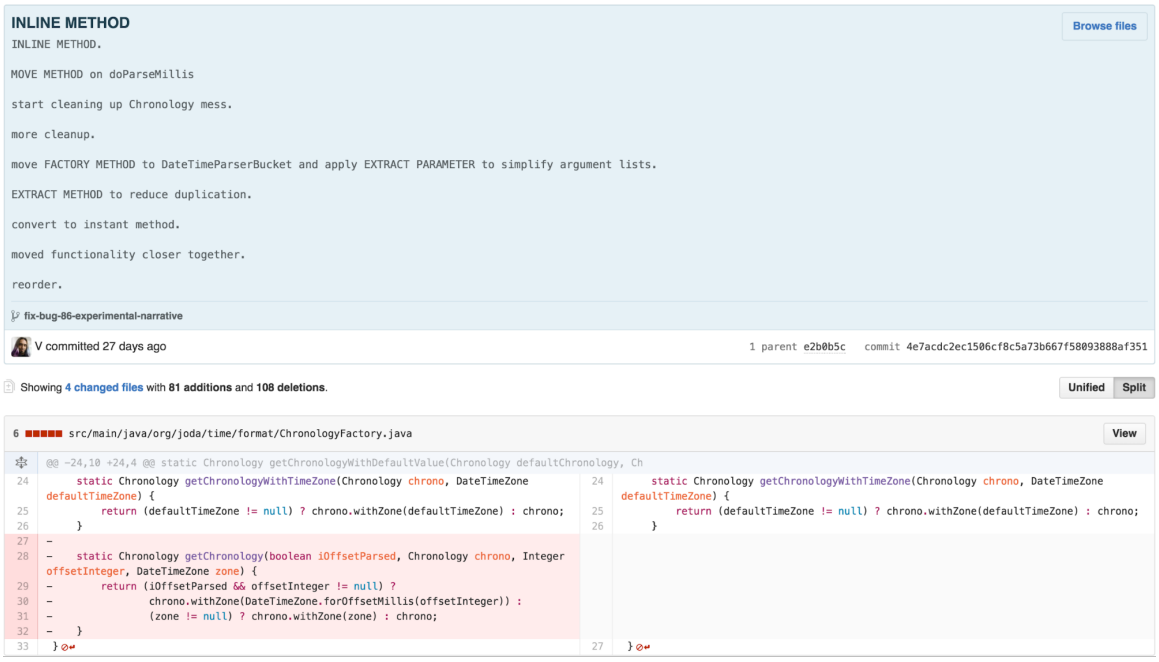


Figure C.76: insert caption

Figure C.77: insert caption

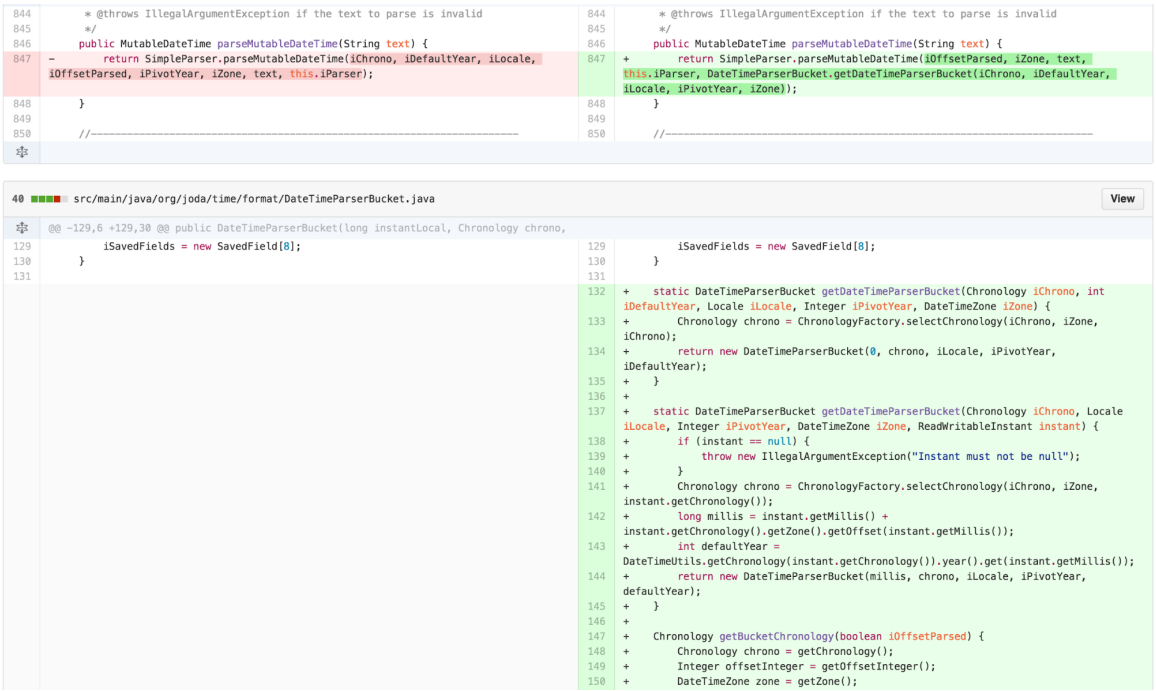


Figure C.78: insert caption

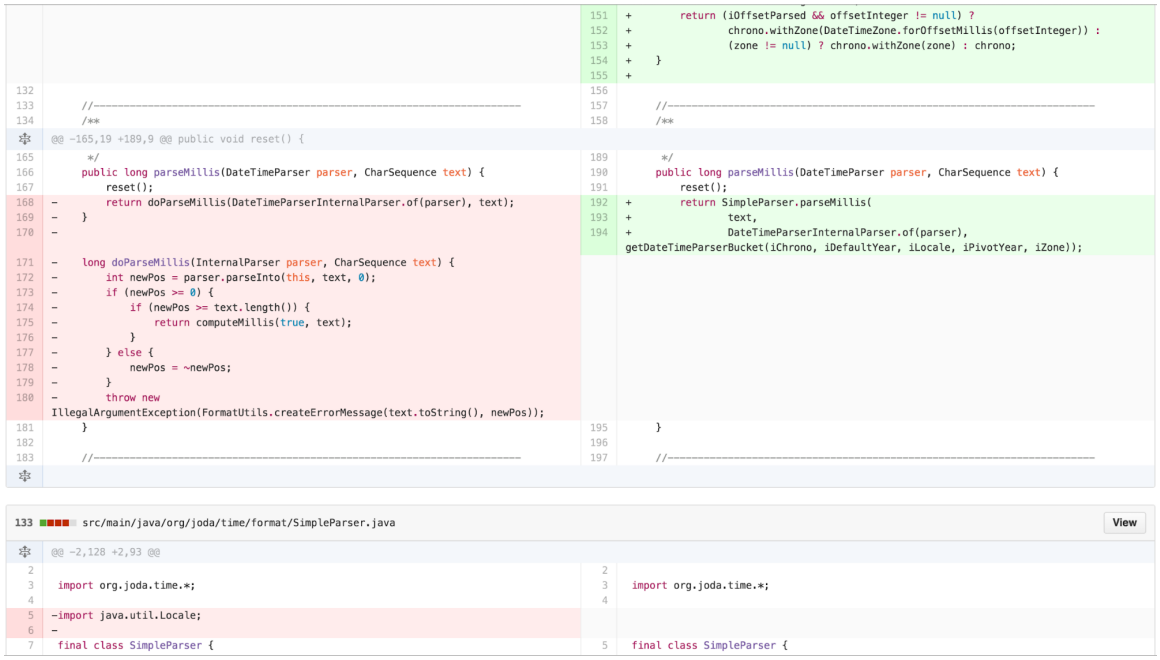


Figure C.79: insert caption

Figure C.80: insert caption

Figure C.81: insert caption

Figure C.82: insert caption

Figure C.83: insert caption

Figure C.84: insert caption

Now there was obvious duplication and an outline structure in the body of the `parse*` methods. I was now able to apply EXTRACT METHOD to identify the points that varied.

apply EXTRACT METHOD to identify duplication in methods. Browse files

fix-bug-86-experimental-narrative

V committed 13 days ago 1 parent 4e7acdc commit f78249f0a9017519b3b10fbe58ea69fdf1217cad

Showing 1 changed file with 37 additions and 21 deletions. Unified Split

58 src/main/java/org/joda/time/format/SimpleParser.java View

```
@@ -20,14 +20,18 @@ static long parseMillis(CharSequence text, InternalParser parser, DateTimeParser
20 20          int newPos = parser.parseInto(bucket, text, 0);
21 21          if (newPos >= 0) {
22 22              if (newPos >= text.length()) {
23    -                 return bucket.computeMillis(true, text);
   23 +                 return getMillis(text, bucket);
24 24              }
25 25          } else {
26 26              newPos = ~newPos;
27 27          }
28 28          throw new IllegalArgumentException(FormatUtils.createErrorMessage(text.toString(), newPos));
29 29      }
30 30
   31 +    private static long getMillis(CharSequence text, DateTimeParserBucket bucket) {
   32 +        return bucket.computeMillis(true, text);
   33 +    }
   34 +
31 35      static LocalDateTime parseLocalDateTime(CharSequence text, InternalParser parser, DateTimeParserBucket bucket) {
32 36          if (parser == null) {
33 37              throw new UnsupportedOperationException("Parsing not supported");
@@ -36,42 +40,50 @@ static LocalDateTime parseLocalDateTime(CharSequence text, InternalParser parser
36 40          int newPos = parser.parseInto(bucket, text, 0);
37 41          if (newPos >= 0) {
38 42              if (newPos >= text.length()) {
39    -                 long millis = bucket.computeMillis(true, text);
40    -                 Chronology chrono = bucket.getChronology();
41    -                 if (bucket.getOffsetInteger() != null) {  // treat withOffsetParsed() as being true
42    -                     int parsedOffset = bucket.getOffsetInteger();
43    -                     DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
44    -                     chrono = chrono.withZone(parsedZone);
45    -                 } else if (bucket.getZone() != null) {
46    -                     chrono = chrono.withZone(bucket.getZone());
47    -                 }
48    -                 return new LocalDateTime(millis, chrono);
```

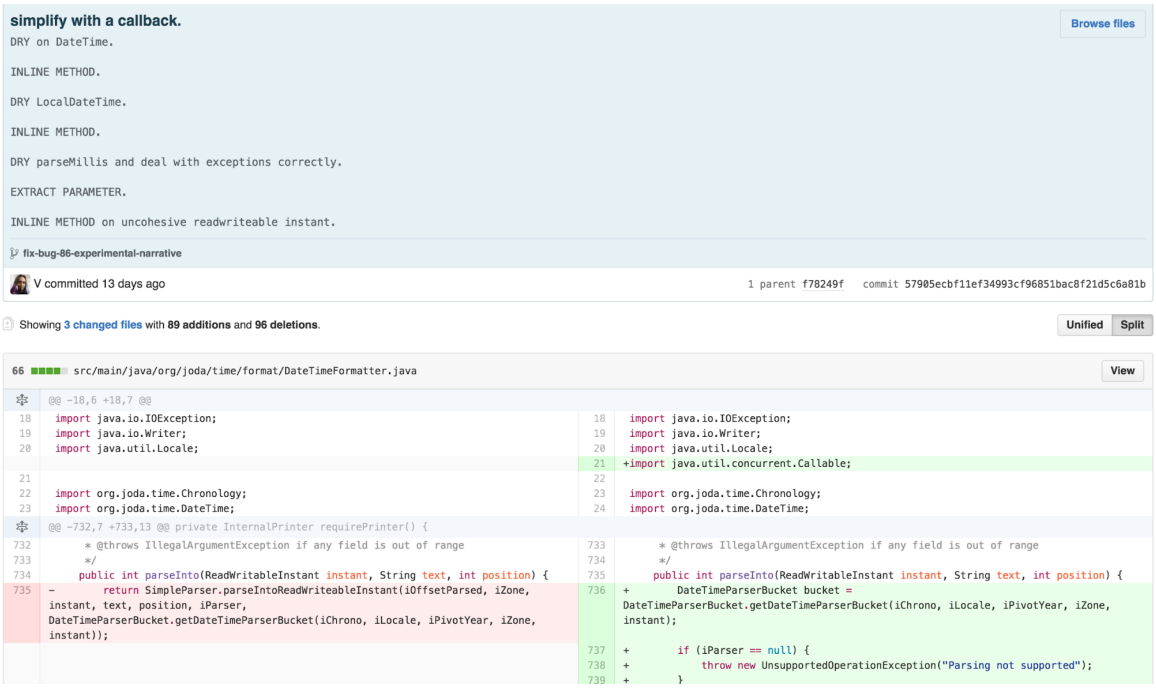
Figure C.85: insert caption

```
                +                return getLocalDateTime(text, bucket);
49  44              }
50  45          } else {
51  46              newPos = ~newPos;
52  47          }
53  48          throw new IllegalArgumentException(FormatUtils.createErrorMessage(text.toString(), newPos));
54  49      }
55  50
            +    private static LocalDateTime getLocalDateTime(CharSequence text, DateTimeParserBucket bucket) {
            +        long millis = getMillis(text, bucket);
            +        Chronology chrono = bucket.getChronology();
            +        if (bucket.getOffsetInteger() != null) {  // treat withOffsetParsed() as being true
            +            int parsedOffset = bucket.getOffsetInteger();
            +            DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
            +            chrono = chrono.withZone(parsedZone);
            +        } else if (bucket.getZone() != null) {
            +            chrono = chrono.withZone(bucket.getZone());
            +        }
            +        return new LocalDateTime(millis, chrono);
            +    }
            +
56  64      static DateTime parseDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, InternalParser parser, DateTi
57  65          if (parser == null) {
58  66              throw new UnsupportedOperationException("Parsing not supported");
59  67          }
60  68          int newPos = parser.parseInto(bucket, text, 0);
61  69          if (newPos >= 0) {
62  70              if (newPos >= text.length()) {
63      -                DateTime dt = new DateTime(bucket.computeMillis(true, text), bucket.getBucketChronology(iOffsetParsed))
64      -                if (iZone != null) {
65      -                    dt = dt.withZone(iZone);
66      -                }
67      -                return dt;
            +                return getDateTime(iOffsetParsed, iZone, text, bucket);
68  72              }
69  73          } else {
70  74              newPos = ~newPos;
71  75          }
72  76          throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
73  77      }
74  78
            +    private static DateTime getDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, DateTimeParserBucket bu
            +        DateTime dt = new DateTime(bucket.computeMillis(true, text), bucket.getBucketChronology(iOffsetParsed));
            +        if (iZone != null) {
            +            dt = dt.withZone(iZone);
            +        }
```

Figure C.86: insert caption

```
+         return dt;
+     }
+
      static MutableDateTime parseMutableDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, InternalParser
          if (parser == null) {
@@ -80,15 +92,19 @@ static MutableDateTime parseMutableDateTime(boolean iOffsetParsed, DateTimeZone
          int newPos = parser.parseInto(bucket, text, 0);
          if (newPos >= 0) {
              if (newPos >= text.length()) {
-                 MutableDateTime dt = new MutableDateTime(bucket.computeMillis(true, text), bucket.getBucketChronology(i
-                 if (iZone != null) {
-                     dt.setZone(iZone);
-                 }
-                 return dt;
+                 return getMutableDateTime(iOffsetParsed, iZone, text, bucket);
              }
          } else {
              newPos = ~newPos;
          }
          throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
      }
+
+     private static MutableDateTime getMutableDateTime(boolean iOffsetParsed, DateTimeZone iZone, String text, DateTimeP
+         MutableDateTime dt = new MutableDateTime(bucket.computeMillis(true, text), bucket.getBucketChronology(iOffsetPa
+         if (iZone != null) {
+             dt.setZone(iZone);
+         }
+         return dt;
+     }
  }
```

Figure C.87: insert caption



simplify with a callback.

Browse files

DRY on DateTime.

INLINE METHOD.

DRY LocalDateTime.

INLINE METHOD.

DRY parseMillis and deal with exceptions correctly.

EXTRACT PARAMETER.

INLINE METHOD on uncohesive readwriteable instant.

fix-bug-86-experimental-narrative

V committed 13 days ago — 1 parent f78249f — commit 57905ecbf11ef34993cf96851bac8f21d5c6a81b

Showing 3 changed files with 89 additions and 96 deletions. — Unified | Split

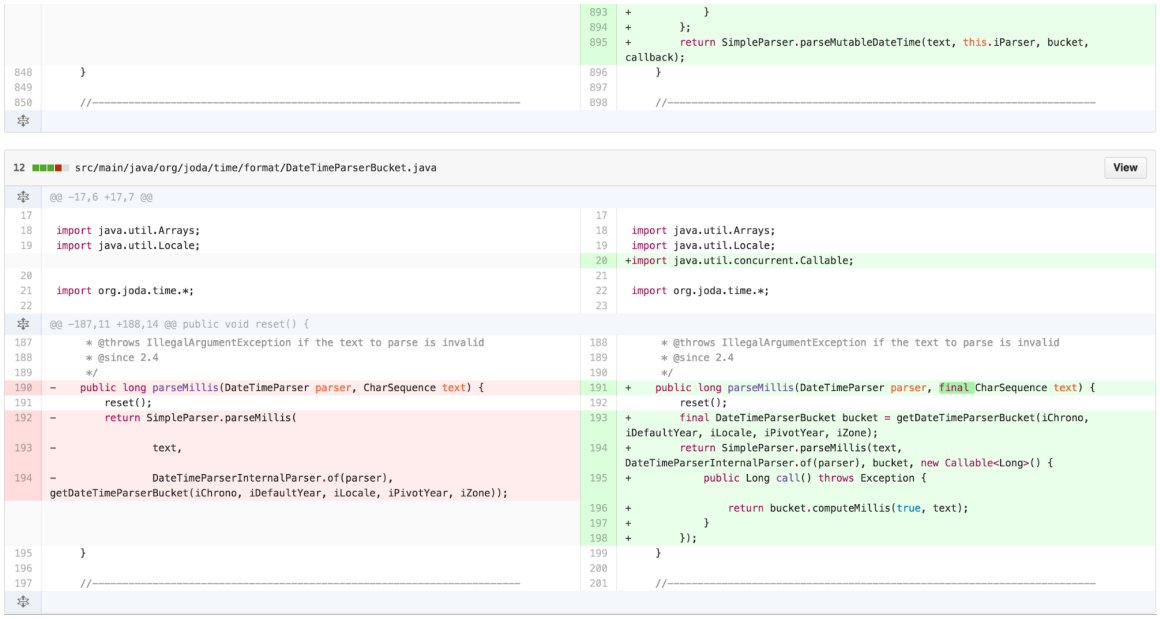
66 src/main/java/org/joda/time/format/DateTimeFormatter.java — View

```
@@ -18,6 +18,7 @@
 import java.io.IOException;
 import java.io.Writer;
 import java.util.Locale;
+import java.util.concurrent.Callable;

 import org.joda.time.Chronology;
 import org.joda.time.DateTime;
@@ -732,7 +733,13 @@ private InternalPrinter requirePrinter() {
      * @throws IllegalArgumentException if any field is out of range
      */
     public int parseInto(ReadWritableInstant instant, String text, int position) {
-        return SimpleParser.parseIntoReadWriteableInstant(iOffsetParsed, iZone,
-                instant, text, position, iParser,
-                DateTimeParserBucket.getDateTimeParserBucket(iChrono, iLocale, iPivotYear, iZone,
-                instant));
+        DateTimeParserBucket bucket =
+                DateTimeParserBucket.getDateTimeParserBucket(iChrono, iLocale, iPivotYear, iZone,
+                instant);
+
+        if (iParser == null) {
+            throw new UnsupportedOperationException("Parsing not supported");
+        }
```

Figure C.88: insert caption

Figure C.89: insert caption

Figure C.90: insert caption

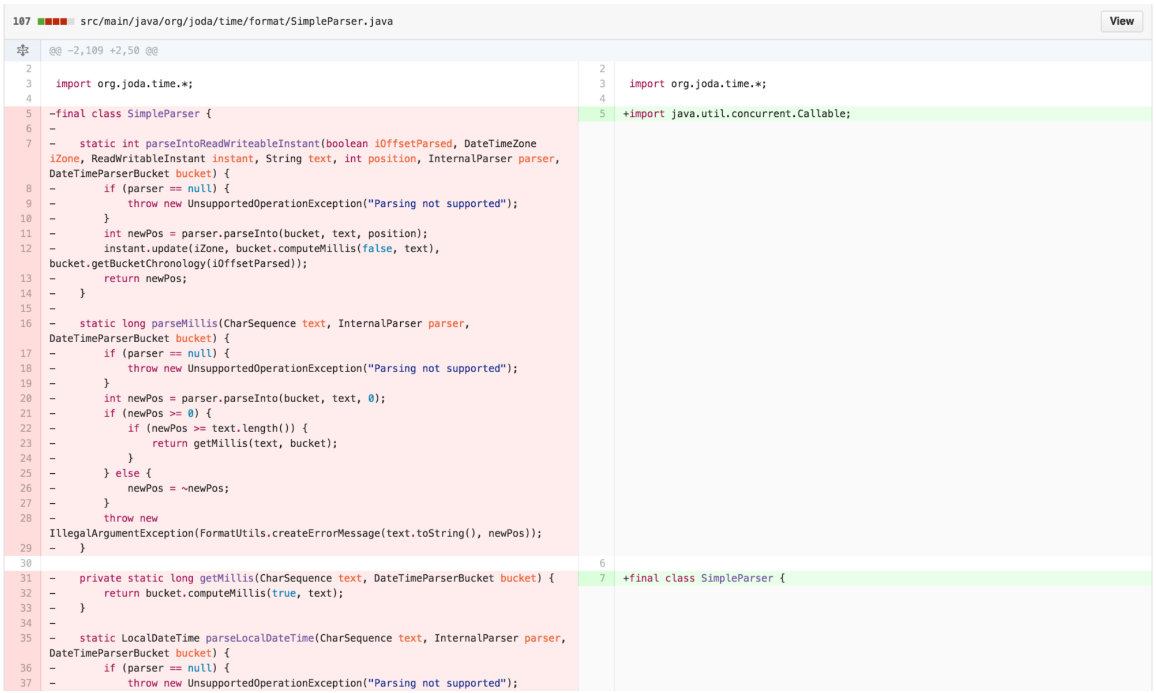


```
                                                                                +            }
                                                                                +        };
                                                                                +        return SimpleParser.parseMutableDateTime(text, this.iParser, bucket,
                                                                                callback);
     }                                                                               }

     //-----------------------------------------------------------------------   //-----------------------------------------------------------------------
```

12 src/main/java/org/joda/time/format/DateTimeParserBucket.java — View

```
@@ -17,6 +17,7 @@
 import java.util.Arrays;
 import java.util.Locale;
                                                                                 +import java.util.concurrent.Callable;

 import org.joda.time.*;

@@ -187,11 +188,14 @@ public void reset() {
      * @throws IllegalArgumentException if the text to parse is invalid
      * @since 2.4
      */
-    public long parseMillis(DateTimeParser parser, CharSequence text) {
+    public long parseMillis(DateTimeParser parser, final CharSequence text) {
         reset();
-        return SimpleParser.parseMillis(
-                text,
-                DateTimeParserInternalParser.of(parser),
getDateTimeParserBucket(iChrono, iDefaultYear, iLocale, iPivotYear, iZone));
+        final DateTimeParserBucket bucket = getDateTimeParserBucket(iChrono,
iDefaultYear, iLocale, iPivotYear, iZone);
+        return SimpleParser.parseMillis(text,
DateTimeParserInternalParser.of(parser), bucket, new Callable<Long>() {
+            public Long call() throws Exception {
+                return bucket.computeMillis(true, text);
+            }
+        });
     }

     //-----------------------------------------------------------------------
```

Figure C.91: insert caption



107 src/main/java/org/joda/time/format/SimpleParser.java — View

```
@@ -2,109 +2,50 @@
 import org.joda.time.*;
-final class SimpleParser {
-
-    static int parseIntoReadWriteableInstant(boolean iOffsetParsed, DateTimeZone
iZone, ReadWritableInstant instant, String text, int position, InternalParser parser,
DateTimeParserBucket bucket) {
-        if (parser == null) {
-            throw new UnsupportedOperationException("Parsing not supported");
-        }
-        int newPos = parser.parseInto(bucket, text, position);
-        instant.update(iZone, bucket.computeMillis(false, text),
bucket.getBucketChronology(iOffsetParsed));
-        return newPos;
-    }
-
-    static long parseMillis(CharSequence text, InternalParser parser,
DateTimeParserBucket bucket) {
-        if (parser == null) {
-            throw new UnsupportedOperationException("Parsing not supported");
-        }
-        int newPos = parser.parseInto(bucket, text, 0);
-        if (newPos >= 0) {
-            if (newPos >= text.length()) {
-                return getMillis(text, bucket);
-            }
-        } else {
-            newPos = ~newPos;
-        }
-        throw new
IllegalArgumentException(FormatUtils.createErrorMessage(text.toString(), newPos));
-    }
-
-    private static long getMillis(CharSequence text, DateTimeParserBucket bucket) {
-        return bucket.computeMillis(true, text);
-    }
-
-    static LocalDateTime parseLocalDateTime(CharSequence text, InternalParser parser,
DateTimeParserBucket bucket) {
-        if (parser == null) {
-            throw new UnsupportedOperationException("Parsing not supported");
+import java.util.concurrent.Callable;
+final class SimpleParser {
```

Figure C.92: insert caption

Figure C.93: insert caption

Figure C.94: insert caption

I then had a couple of options. The traditional TEMPLATE METHOD approach

would require refactoring the `SimpleParser` to be a fully-fledged object instead of a utility class full of static methods. This approach required some intensive structural modifications, though is fairly straightforward. However, it is an approach most optimally suited for a traditional object-oriented programming paradigm.

Functional Programming is experiencing a revival in the software engineering community. As Moore's Law runs into physical nanoscale limitations, chip manufacturers have sought to improve speeds by increasing the number of cores on a chip and promoting parallel computing. Pure, stateless functions are easier to parallelize. Hence, languages like Ruby, Python, Javascript, Scala, and Clojure began supporting functional paradigms. Eventually, both Java 8 and C++ 11 added functional support features. Many young programmers learning today may have more experience with functional idioms than in the past.

One common functional idiom is the notion of a callback function. When using callbacks, one passes a function to be invoked to another function. The other function is then able to execute and invoke the passed function at its leisure. I explored this approach with a subsequent set of refactoring.

With some slight caveats to handle the difference in exception handling introduced by Java's Callable, this approach compiles and executes successfully. The question is, is it simpler?

Callbacks are often argued to make code harder to understand. Some argue that they can be used effectively and comprehensibly if the number is limited. There is no data comparing the cognitive load induced by following such an approach versus traditional OOP. This seems like ground rife for future research.

Returning our attention to simplifying the usage patterns, an obvious opportunity to reduce the Split Attention Effect is to use constructor chaining when the initialization semantics are the same and the only difference is a variable number of arguments. Hence, we can simplify one of the `DateTimeFormatter` constructors as follows.

52 src/main/java/org/joda/time/format/DateTimeFormatter.java View

@@ -127,15 +127,13 @@ public DateTimeFormatter(

```
127      */                                                  127      */
128     DateTimeFormatter(                                   128     DateTimeFormatter(
129             InternalPrinter printer, InternalParser parser) {   129             InternalPrinter printer, InternalParser parser) {
130 -       super();                                         130 +       this(printer, parser,
131 -       iPrinter = printer;                              131 +            null,
132 -       iParser = parser;                                132 +            false,
133 -       iLocale = null;                                  133 +            null,
134 -       iOffsetParsed = false;                           134 +            null,
135 -       iChrono = null;                                  135 +            null,
136 -       iZone = null;                                    136 +            2000);
137 -       iPivotYear = null;
138 -       iDefaultYear = 2000;
139     }                                                    137     }
140                                                          138
141     /**                                                  139     /**
```

Figure C.95: insert caption

We can also significantly simplify the signatures of `SimpleParser` when we realize that most of its arguments are pulled from `DateTimeFormatter`.

Figure C.96: insert caption

This significantly simplifies `DateTimeParserBucket`:

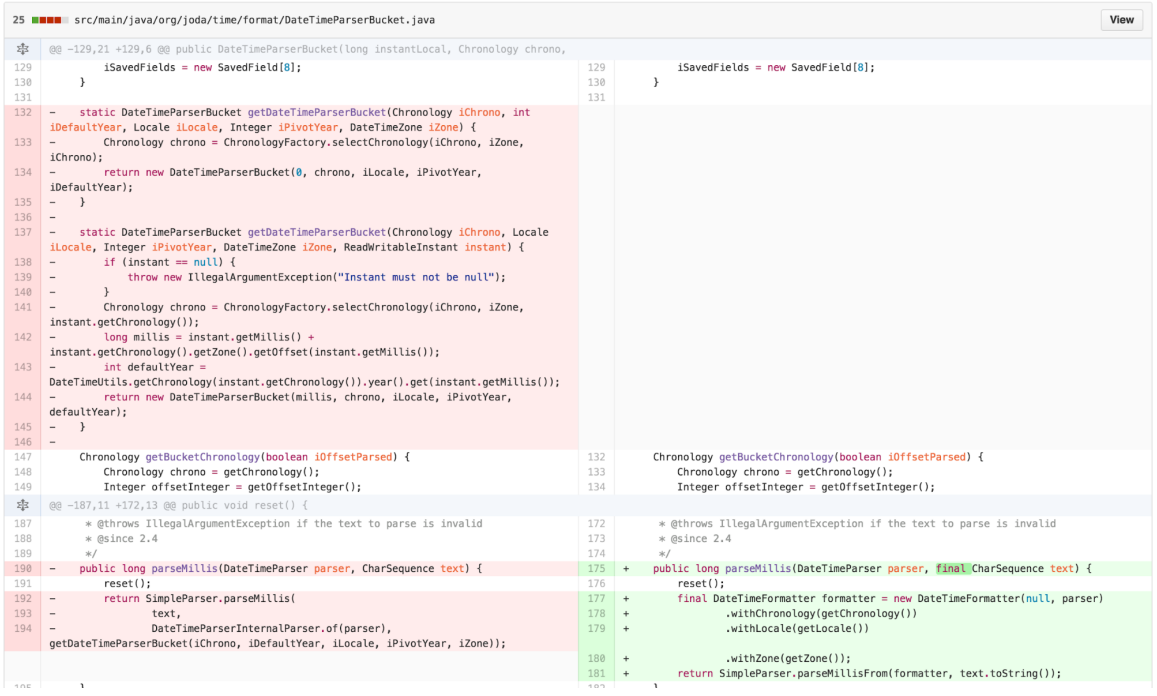


Figure C.97: insert caption

`SimpleParser` then provides a FACADE that wraps its callback generation and generalized `getResult` method.

159 src/main/java/org/joda/time/format/SimpleParser.java View

```
@@ -2,109 +2,112 @@

 import org.joda.time.*;

-final class SimpleParser {

-    static int parseIntoReadWriteableInstant(boolean iOffsetParsed, DateTimeZone iZone, ReadWritableInstant instant, String text, int position, InternalParser parser, DateTimeParserBucket bucket) {
-        if (parser == null) {
-            throw new UnsupportedOperationException("Parsing not supported");
-        }
-        int newPos = parser.parseInto(bucket, text, position);
-        instant.update(iZone, bucket.computeMillis(false, text), bucket.getBucketChronology(iOffsetParsed));
-        return newPos;
-    }

-    static long parseMillis(CharSequence text, InternalParser parser, DateTimeParserBucket bucket) {
-        if (parser == null) {
-            throw new UnsupportedOperationException("Parsing not supported");
-        }
-        int newPos = parser.parseInto(bucket, text, 0);
-        if (newPos >= 0) {
-            if (newPos >= text.length()) {
-                return getMillis(text, bucket);



            }
-        } else {
-            newPos = ~newPos;
-        }
-        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text.toString(), newPos));
    }

-    private static long getMillis(CharSequence text, DateTimeParserBucket bucket) {

-        return bucket.computeMillis(true, text);
-    }
-

 import org.joda.time.*;

+import java.util.Locale;
+import java.util.concurrent.Callable;

+final class SimpleParser {

+    static long parseMillisFrom(DateTimeFormatter dateTimeFormatter, final String text) {
+        final DateTimeParserBucket bucket = getDateTimeParserBucket(
+                dateTimeFormatter.getChronology(),
+                dateTimeFormatter.getDefaultYear(),
+                dateTimeFormatter.getLocale(),
+                dateTimeFormatter.getPivotYear(),
+                dateTimeFormatter.getZone(), 0);
+        final Callable<Long> callback = new Callable<Long>() {
+            public Long call() throws Exception {
+                return bucket.computeMillis(true, text);
            }
+        };
+        return getResult(text, dateTimeFormatter.getParser0(), bucket, callback);
    }

+    private static <T> T getResult(String text, InternalParser parser, DateTimeParserBucket bucket, Callable<T> callable) {
```

Figure C.98: insert caption

```
-    static LocalDateTime parseLocalDateTime(CharSequence text, InternalParser parser, DateTimeParserBucket bucket) {
         if (parser == null) {
             throw new UnsupportedOperationException("Parsing not supported");
         }
-
         int newPos = parser.parseInto(bucket, text, 0);
         if (newPos >= 0) {
             if (newPos >= text.length()) {
-                return getLocalDateTime(text, bucket);



             }
         } else {
             newPos = ~newPos;
         }
-        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text.toString(), newPos));
-    }
-
-    private static LocalDateTime getLocalDateTime(CharSequence text, DateTimeParserBucket bucket) {
-        long millis = getMillis(text, bucket);
-        Chronology chrono = bucket.getChronology();
-        if (bucket.getOffsetInteger() != null) {  // treat withOffsetParsed() as being true
-            int parsedOffset = bucket.getOffsetInteger();
-            DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
-            chrono = chrono.withZone(parsedZone);
-        } else if (bucket.getZone() != null) {
-            chrono = chrono.withZone(bucket.getZone());
-        }
-        return new LocalDateTime(millis, chrono);
     }

         if (parser == null) {
             throw new UnsupportedOperationException("Parsing not supported");
         }
         int newPos = parser.parseInto(bucket, text, 0);
         if (newPos >= 0) {
             if (newPos >= text.length()) {
+                try {
+                    return callable.call();
+                } catch (Exception e) {
+                    if (e instanceof RuntimeException) {
+                        throw (RuntimeException)e;
+                    }
+                }
             }
         } else {
             newPos = ~newPos;
         }
+        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
     }
```

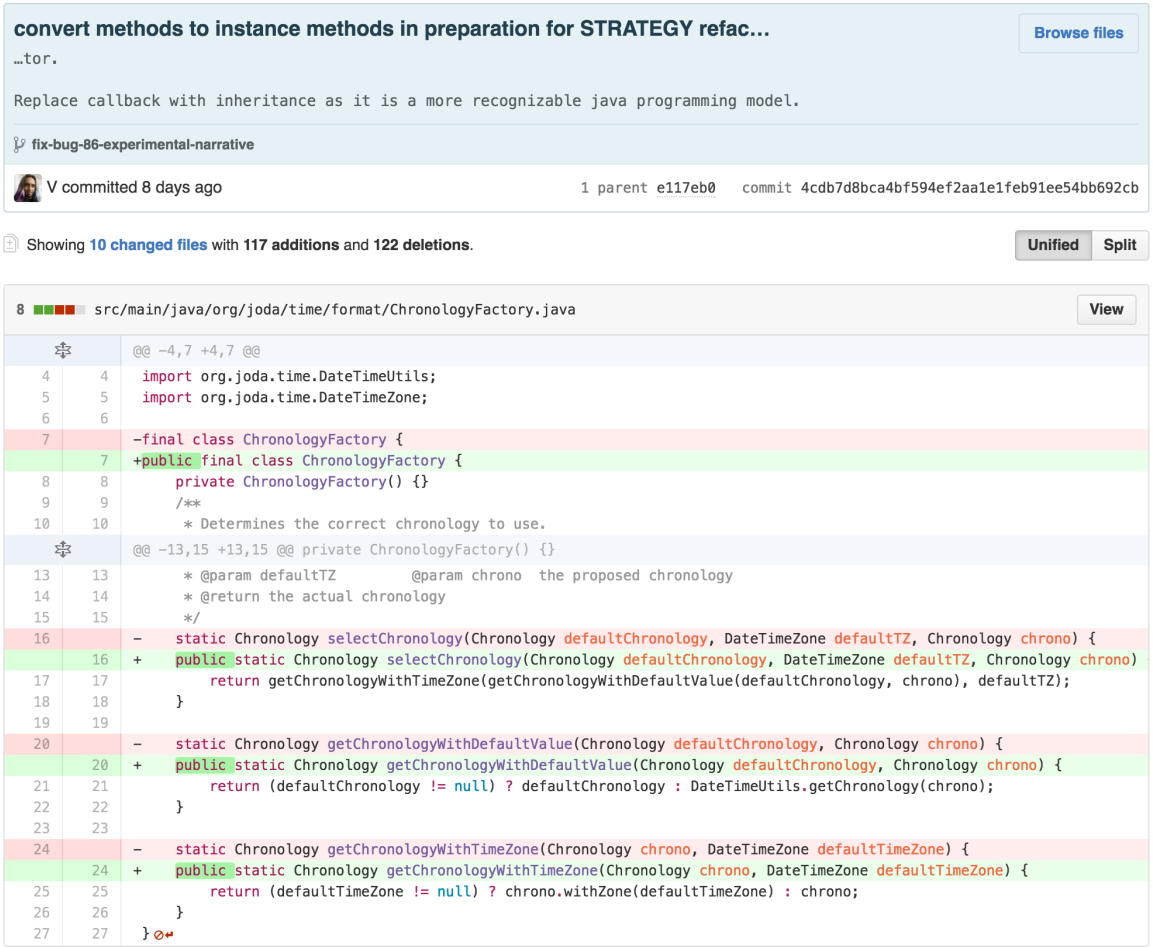
Figure C.99: insert caption

Figure C.100: insert caption

Figure C.101: insert caption

At this point, we are ready to apply our instance transformation to form a TEMPLATE METHOD and have individual STRATEGY implementations for each return

type.



convert methods to instance methods in preparation for STRATEGY refac... Browse files

…tor.

Replace callback with inheritance as it is a more recognizable java programming model.

fix-bug-86-experimental-narrative

V committed 8 days ago 1 parent e117eb0 commit 4cdb7d8bca4bf594ef2aa1e1feb91ee54bb692cb

Showing 10 changed files with 117 additions and 122 deletions. Unified Split

8 src/main/java/org/joda/time/format/ChronologyFactory.java View

```
@@ -4,7 +4,7 @@
 import org.joda.time.DateTimeUtils;
 import org.joda.time.DateTimeZone;

-final class ChronologyFactory {
+public final class ChronologyFactory {
     private ChronologyFactory() {}
     /**
      * Determines the correct chronology to use.
@@ -13,15 +13,15 @@ private ChronologyFactory() {}
      * @param defaultTZ         @param chrono  the proposed chronology
      * @return the actual chronology
      */
-    static Chronology selectChronology(Chronology defaultChronology, DateTimeZone defaultTZ, Chronology chrono) {
+    public static Chronology selectChronology(Chronology defaultChronology, DateTimeZone defaultTZ, Chronology chrono)
         return getChronologyWithTimeZone(getChronologyWithDefaultValue(defaultChronology, chrono), defaultTZ);
     }

-    static Chronology getChronologyWithDefaultValue(Chronology defaultChronology, Chronology chrono) {
+    public static Chronology getChronologyWithDefaultValue(Chronology defaultChronology, Chronology chrono) {
         return (defaultChronology != null) ? defaultChronology : DateTimeUtils.getChronology(chrono);
     }

-    static Chronology getChronologyWithTimeZone(Chronology chrono, DateTimeZone defaultTimeZone) {
+    public static Chronology getChronologyWithTimeZone(Chronology chrono, DateTimeZone defaultTimeZone) {
         return (defaultTimeZone != null) ? chrono.withZone(defaultTimeZone) : chrono;
     }
 }
```

Figure C.102: insert caption

8 src/main/java/org/joda/time/format/DateTimeFormatter.java View

```
@@ -761,7 +761,7 @@ public int parseInto(ReadWritableInstant instant, String text, int position) {
761 761      * @throws IllegalArgumentException if the text to parse is invalid
762 762      */
763 763     public long parseMillis(final String text) {
764     -        return SimpleParser.parseMillisFrom(this, text);
    764 +        return new MillsecondParsingStrategy(this).parse(text);
765 765     }
766 766
767 767     /**
@@ -815,7 +815,7 @@ public LocalTime parseLocalTime(String text) {
815 815      * @since 2.0
816 816      */
817 817     public LocalDateTime parseLocalDateTime(final String text) {
818     -        return SimpleParser.parseIntoLocalDateTime(this, text);
    818 +        return new LocalDateTimeParsingStrategy(this).parse(text);
819 819     }
820 820
821 821     /**
@@ -836,7 +836,7 @@ public LocalDateTime parseLocalDateTime(final String text) {
836 836      * @throws IllegalArgumentException if the text to parse is invalid
837 837      */
838 838     public DateTime parseDateTime(final String text) {
839     -        return SimpleParser.getDateTime(this, text);
    839 +        return new DateTimeParsingStrategy(this).parse(text);
840 840     }
841 841
842 842     /**
@@ -857,6 +857,6 @@ public DateTime parseDateTime(final String text) {
857 857      * @throws IllegalArgumentException if the text to parse is invalid
858 858      */
859 859     public MutableDateTime parseMutableDateTime(final String text) {
860     -        return SimpleParser.getMutableDateTime(this, text);
    860 +        return new MutableDateTimeParsingStrategy(this).parse(text);
861 861     }
862 862 }
```

Figure C.103: insert caption

2 src/main/java/org/joda/time/format/DateTimeParserBucket.java View

```
@@ -178,7 +178,7 @@ public long parseMillis(DateTimeParser parser, final CharSequence text) {
178 178                 .withChronology(getChronology())
179 179                 .withLocale(getLocale())
180 180                 .withZone(getZone());
181     -       return SimpleParser.parseMillisFrom(formatter, text.toString());
    181 +       return new MillsecondParsingStrategy(formatter).parse(text.toString());
182 182     }
183 183
184 184     //-----------------------------------------------------------------------
```

17 src/main/java/org/joda/time/format/DateTimeParsingStrategy.java View

```
@@ -0,0 +1,17 @@
1  +package org.joda.time.format;
2  +
3  +import org.joda.time.DateTime;
4  +
5  +final class DateTimeParsingStrategy extends FormatterParsingStrategy<DateTime> {
6  +    DateTimeParsingStrategy(DateTimeFormatter formatter) {
7  +        super(formatter);
8  +    }
9  +
10 +    protected DateTime doParse(final CharSequence text) {
11 +        DateTime dt = new DateTime(bucket.computeMillis(true, text), bucket.getBucketChronology(formatter.isOffsetParse
12 +        if (formatter.getZone() != null) {
13 +            dt = dt.withZone(formatter.getZone());
14 +        }
15 +        return dt;
16 +    }
17 +}
```

Figure C.104: insert caption

31 ■■■■■ src/main/java/org/joda/time/format/FormatterParsingStrategy.java View

```
@@ -0,0 +1,31 @@
+package org.joda.time.format;
+
+import org.joda.time.Chronology;
+
+abstract class FormatterParsingStrategy<T> implements ParsingStrategy<T> {
+    protected final DateTimeFormatter formatter;
+    protected final DateTimeParserBucket bucket;
+
+    FormatterParsingStrategy(final DateTimeFormatter formatter) {
+        this.formatter = formatter;
+        Chronology chrono = ChronologyFactory.selectChronology(formatter.getChronology(), formatter.getZone(), formatte
+        this.bucket = new DateTimeParserBucket((long) 0, chrono, formatter.getLocale(), formatter.getPivotYear(), forma
+    }
+
+    public T parse(CharSequence text) {
+        if (formatter.getParser0() == null) {
+            throw new UnsupportedOperationException("Parsing not supported");
+        }
+        int newPos = formatter.getParser0().parseInto(bucket, text, 0);
+        if (newPos >= 0) {
+            if (newPos >= text.length()) {
+                return doParse(text);
+            }
+        } else {
+            newPos = ~newPos;
+        }
+        throw new IllegalArgumentException(FormatUtils.createErrorMessage(text.toString(), newPos));
+    }
+
+    protected abstract T doParse(CharSequence text);
+}
```

Figure C.105: insert caption

25 ■■■■■ src/main/java/org/joda/time/format/LocalDateTimeParsingStrategy.java View

```
@@ -0,0 +1,25 @@
+package org.joda.time.format;
+
+import org.joda.time.Chronology;
+import org.joda.time.DateTimeZone;
+import org.joda.time.LocalDateTime;
+
+final class LocalDateTimeParsingStrategy extends FormatterParsingStrategy<LocalDateTime> {
+
+    LocalDateTimeParsingStrategy(DateTimeFormatter formatter) {
+        super(formatter);
+    }
+
+    protected LocalDateTime doParse(final CharSequence text) {
+        long millis = bucket.computeMillis(true, text);
+        Chronology chrono = bucket.getChronology();
+        if (bucket.getOffsetInteger() != null) {  // treat withOffsetParsed() as being true
+            int parsedOffset = bucket.getOffsetInteger();
+            DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
+            chrono = chrono.withZone(parsedZone);
+        } else if (bucket.getZone() != null) {
+            chrono = chrono.withZone(bucket.getZone());
+        }
+        return new LocalDateTime(millis, chrono);
+    }
+}
```

Figure C.106: insert caption

12 ■■■■■ src/main/java/org/joda/time/format/MillsecondParsingStrategy.java View

```
... ... @@ -0,0 +1,12 @@
1  +package org.joda.time.format;
2  +
3  +final class MillsecondParsingStrategy extends FormatterParsingStrategy<Long> {
4  +
5  +    MillsecondParsingStrategy(DateTimeFormatter formatter) {
6  +        super(formatter);
7  +    }
8  +
9  +    protected Long doParse(final CharSequence text) {
10 +        return bucket.computeMillis(true, text);
11 +    }
12 +}
```

18 ■■■■■ src/main/java/org/joda/time/format/MutableDateTimeParsingStrategy.java View

```
... ... @@ -0,0 +1,18 @@
1  +package org.joda.time.format;
2  +
3  +import org.joda.time.MutableDateTime;
4  +
5  +final class MutableDateTimeParsingStrategy extends FormatterParsingStrategy<MutableDateTime> {
6  +
7  +    MutableDateTimeParsingStrategy(DateTimeFormatter formatter) {
8  +        super(formatter);
9  +    }
10 +
11 +    public MutableDateTime doParse(final CharSequence text) {
12 +        MutableDateTime dt = new MutableDateTime(bucket.computeMillis(true, text), bucket.getBucketChronology(formatter
13 +        if (formatter.getZone() != null) {
14 +            dt.setZone(formatter.getZone());
15 +        }
16 +        return dt;
17 +    }
18 +}
```

Figure C.107: insert caption

5 src/main/java/org/joda/time/format/ParsingStrategy.java View

```
@@ -0,0 +1,5 @@
+package org.joda.time.format;
+
+public interface ParsingStrategy<T> {
+    T parse(CharSequence text);
+}
```

113 src/main/java/org/joda/time/format/SimpleParser.java View

```
@@ -1,113 +0,0 @@
-package org.joda.time.format;
-
-import org.joda.time.*;
-
-import java.util.Locale;
-import java.util.concurrent.Callable;
-
-final class SimpleParser {
-
-    static long parseMillisFrom(DateTimeFormatter dateTimeFormatter, final String text) {
-        final DateTimeParserBucket bucket = getDateTimeParserBucket(
-                dateTimeFormatter.getChronology(),
-                dateTimeFormatter.getDefaultYear(),
-                dateTimeFormatter.getLocale(),
-                dateTimeFormatter.getPivotYear(),
-                dateTimeFormatter.getZone(), 0);
-        final Callable<Long> callback = new Callable<Long>() {
-            public Long call() throws Exception {
-                return bucket.computeMillis(true, text);
-            }
-        };
-        return getResult(text, dateTimeFormatter.getParser0(), bucket, callback);
-    }
-
-    private static <T> T getResult(String text, InternalParser parser, DateTimeParserBucket bucket, Callable<T> callabl
-        if (parser == null) {
-            throw new UnsupportedOperationException("Parsing not supported");
-        }
-        int newPos = parser.parseInto(bucket, text, 0);
-        if (newPos >= 0) {
-            if (newPos >= text.length()) {
-                try {
```

Figure C.108: insert caption

```
-                     return callable.call();
-                 } catch (Exception e) {
-                     if (e instanceof RuntimeException) {
-                         throw (RuntimeException)e;
-                     }
-                 }
-             }
-         } else {
-             newPos = ~newPos;
-         }
-         throw new IllegalArgumentException(FormatUtils.createErrorMessage(text, newPos));
-     }
-
-     static LocalDateTime parseIntoLocalDateTime(DateTimeFormatter dateTimeFormatter, final String text) {
-         final DateTimeParserBucket bucket = getDateTimeParserBucket(
-                 dateTimeFormatter.getChronology(),
-                 dateTimeFormatter.getDefaultYear(),
-                 dateTimeFormatter.getLocale(),
-                 dateTimeFormatter.getPivotYear(),
-                 dateTimeFormatter.getZone(), 0);
-         final Callable<LocalDateTime> callback = new Callable<LocalDateTime>() {
-             public LocalDateTime call() throws Exception {
-                 long millis = bucket.computeMillis(true, text);
-                 Chronology chrono = bucket.getChronology();
-                 if (bucket.getOffsetInteger() != null) {  // treat withOffsetParsed() as being true
-                     int parsedOffset = bucket.getOffsetInteger();
-                     DateTimeZone parsedZone = DateTimeZone.forOffsetMillis(parsedOffset);
-                     chrono = chrono.withZone(parsedZone);
-                 } else if (bucket.getZone() != null) {
-                     chrono = chrono.withZone(bucket.getZone());
-                 }
-                 return new LocalDateTime(millis, chrono);
-             }
-         };
-         return getResult(text, dateTimeFormatter.getParser0(), bucket, callback);
-     }
-
-     static DateTime getDateTime(final DateTimeFormatter dateTimeFormatter, final String text) {
-         final DateTimeParserBucket bucket = getDateTimeParserBucket(
-                 dateTimeFormatter.getChronology(),
-                 dateTimeFormatter.getDefaultYear(),
-                 dateTimeFormatter.getLocale(),
-                 dateTimeFormatter.getPivotYear(),
-                 dateTimeFormatter.getZone(), 0);
-         final Callable<DateTime> callback = new Callable<DateTime>() {
-             public DateTime call() throws Exception {
```

Figure C.109: insert caption

```
79        -                DateTime dt = new DateTime(bucket.computeMillis(true, text), bucket.getBucketChronology(dateTimeFormatt
80        -                if (dateTimeFormatter.getZone() != null) {
81        -                    dt = dt.withZone(dateTimeFormatter.getZone());
82        -                }
83        -                return dt;
84        -            }
85        -        };
86        -        return getResult(text, dateTimeFormatter.getParser0(), bucket, callback);
87        -    }
88        -
89        -    static MutableDateTime getMutableDateTime(final DateTimeFormatter dateTimeFormatter, final String text) {
90        -        final DateTimeParserBucket bucket = getDateTimeParserBucket(
91        -                dateTimeFormatter.getChronology(),
92        -                dateTimeFormatter.getDefaultYear(),
93        -                dateTimeFormatter.getLocale(),
94        -                dateTimeFormatter.getPivotYear(),
95        -                dateTimeFormatter.getZone(), 0);
96        -
97        -        final Callable<MutableDateTime> callback = new Callable<MutableDateTime>() {
98        -            public MutableDateTime call() throws Exception {
99        -                MutableDateTime dt = new MutableDateTime(bucket.computeMillis(true, text), bucket.getBucketChronology(d
100       -                if (dateTimeFormatter.getZone() != null) {
101       -                    dt.setZone(dateTimeFormatter.getZone());
102       -                }
103       -                return dt;
104       -            }
105       -        };
106       -        return getResult(text, dateTimeFormatter.getParser0(), bucket, callback);
107       -    }
108       -
109       -    private static DateTimeParserBucket getDateTimeParserBucket(Chronology iChrono, int iDefaultYear, Locale iLocale, I
110       -        Chronology chrono = ChronologyFactory.selectChronology(iChrono, iZone, iChrono);
111       -        return new DateTimeParserBucket(millis, chrono, iLocale, iPivotYear, iDefaultYear);
112       -    }
113       -}
```

Figure C.110: insert caption

At this point, we've replaced the `SimpleParser` with a `ParsingStrategy` and a `FormatterParsingStrategy` that holds the guts of the `getResult` method, with subclass implementations for each return type.

I now also had a straightforward point to simplify common logic. I also had an interface that I could use to unify the differences between the API design of `parseInto` which mutates the input parameter and returns the resulting position, and the other `parse` methods. This reduces the architectural complexity of `DateTimeFormatter` by making all of the parsing follow a common pattern.

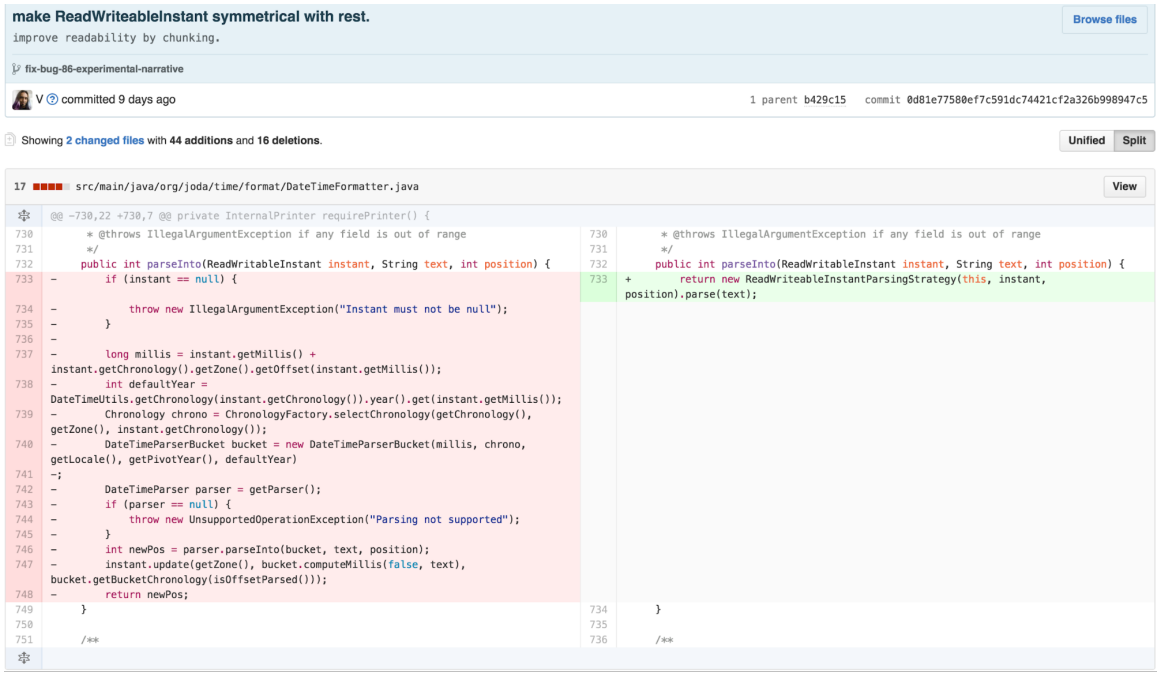


Figure C.111: insert caption

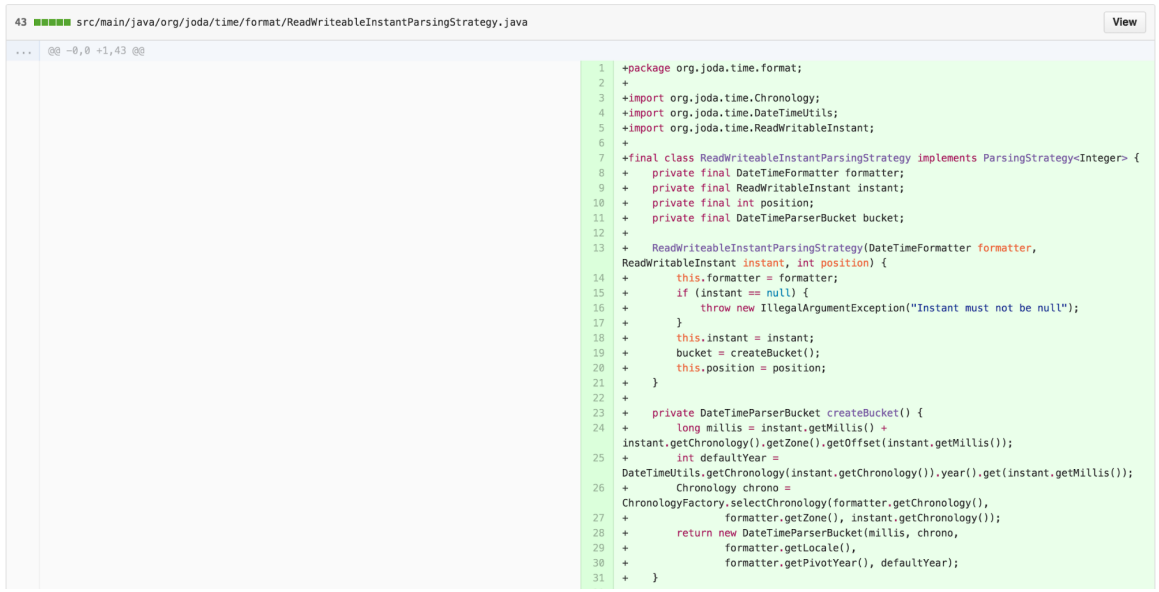
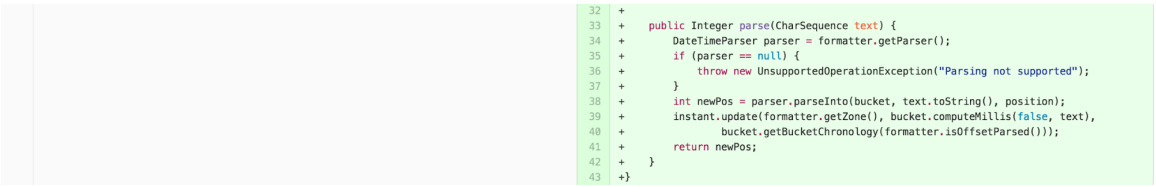


Figure C.112: insert caption



```
+
+    public Integer parse(CharSequence text) {
+        DateTimeParser parser = formatter.getParser();
+        if (parser == null) {
+            throw new UnsupportedOperationException("Parsing not supported");
+        }
+        int newPos = parser.parseInto(bucket, text.toString(), position);
+        instant.update(formatter.getZone(), bucket.computeMillis(false, text),
+                bucket.getBucketChronology(formatter.isOffsetParsed()));
+        return newPos;
+    }
+}
```

Figure C.113: insert caption

When I first tried to make the `ReadWritableInstantParsingStrategy`, I tried directly extending `FormatterParsingStrategy` like the other implementations. When I ran the tests, they failed. Something was different about the way it worked. Luckily, having a separate interface for `ParsingStrategy` from the abstract class implementation of `FormatterParsingStrategy` allowed me to move forward to advance architectural similarity without being blocked. I was able to create the implementation with only slight duplication of work done in the `FormatterParsingStrategy`. When I think about it further, it seems like the primary differences between `ReadWritableInstantParsingStra` and the other implementations is that the starting number of milliseconds for the `DateTimeParserBucket` is not 0, and neither is the position for the `DateTimeParser`. I could've spent some time refactoring `FormatterParsingStrategy` to parametrize it such that `ReadWritableInstantParsingStrategy` could extend it. I decided not to because this code path is unlikely to get explored during the study and the goal of this work was to simplify `DateTimeFormatter`.

C.6.3 Signaling architecture by organization – using packages as chunks

I now had a simpler `DateTimeFormatter` whose parsing methods delegated to implementations of parsing STRATEGY. However, the package org.joda.time.format had grown more imposing by adding all of these new classes. We can use packages to organize classes, again applying the concepts of sequencing and chunking to manage how much context a programmer needs to keep in their head at one time. Hence, we add a new package org.joda.time.format.parsing and move the new classes over.

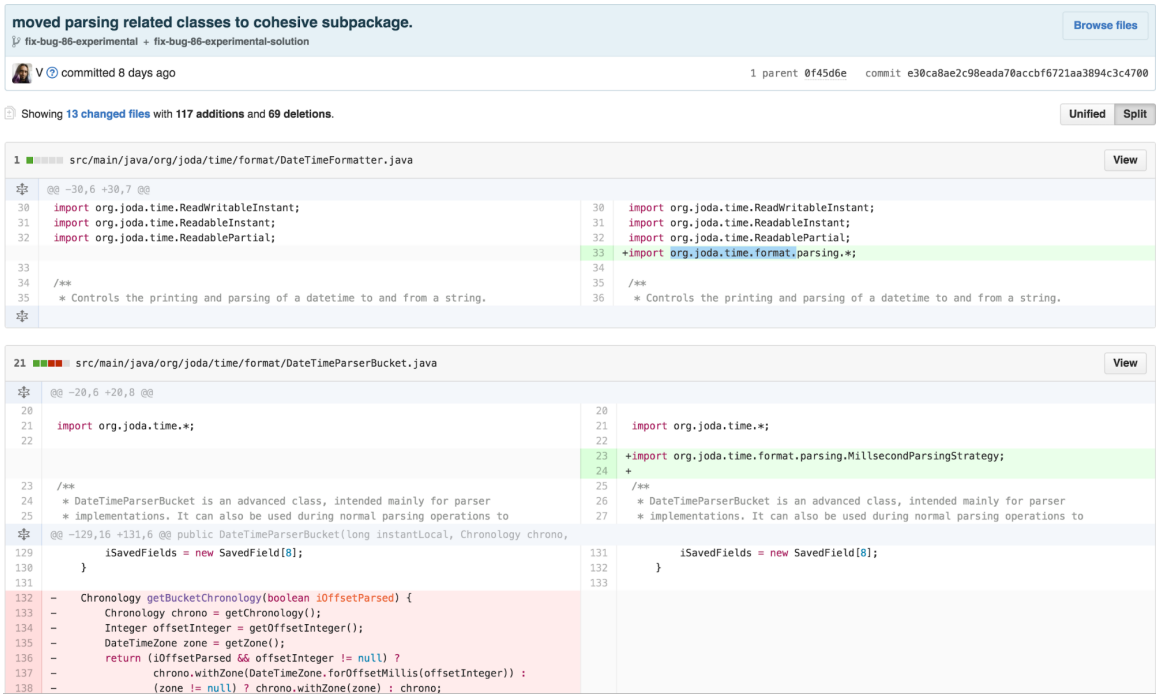


Figure C.114: insert caption

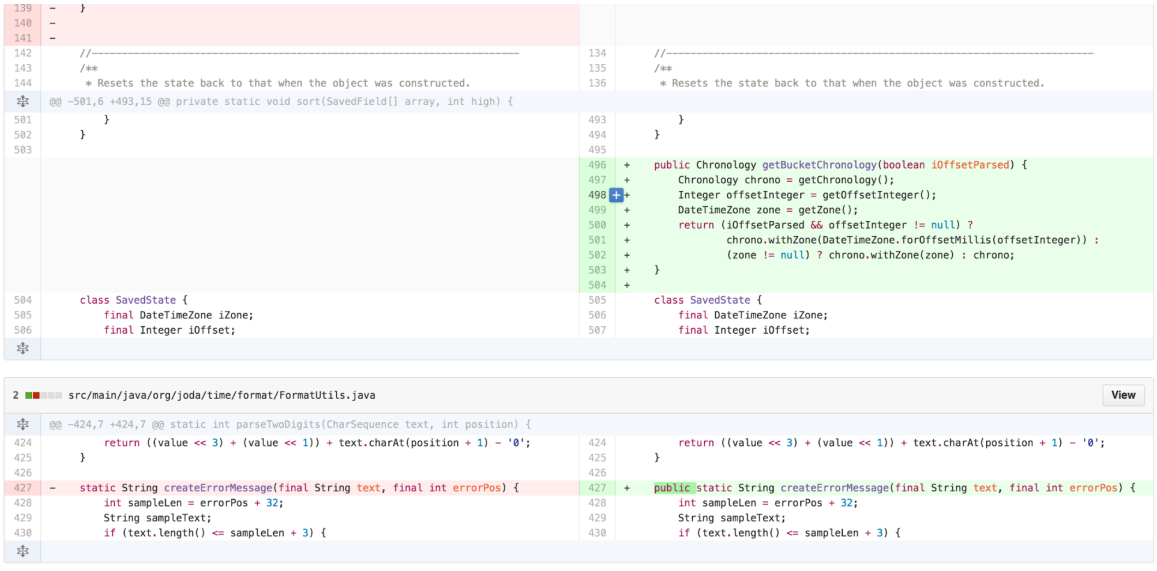


Figure C.115: insert caption

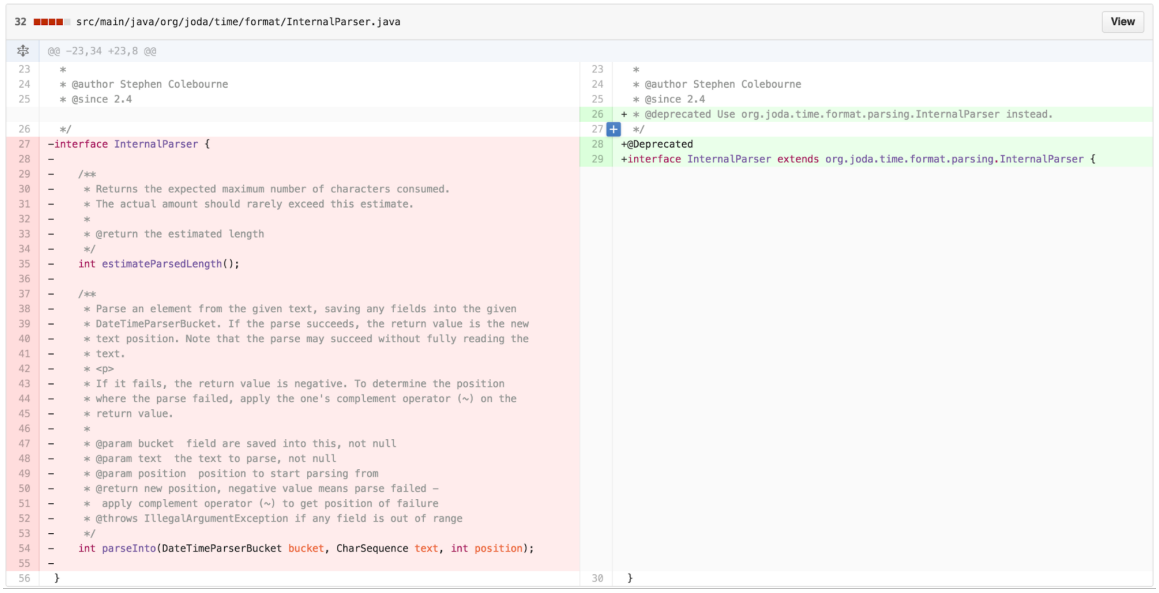


Figure C.116: insert caption

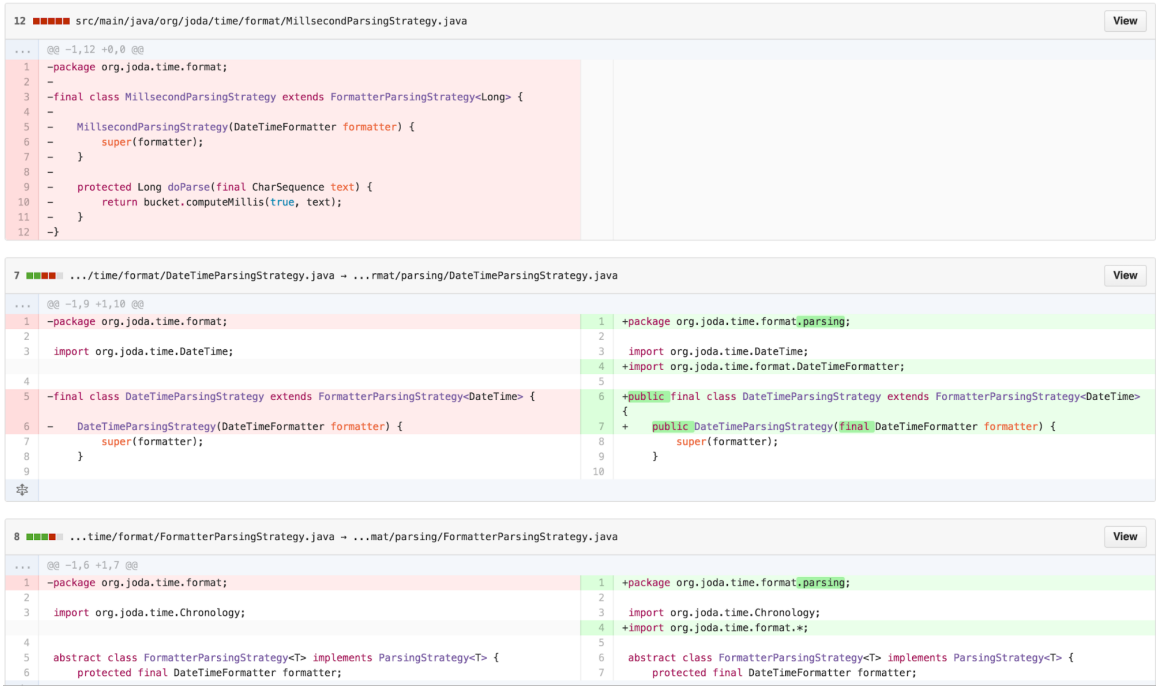


Figure C.117: insert caption

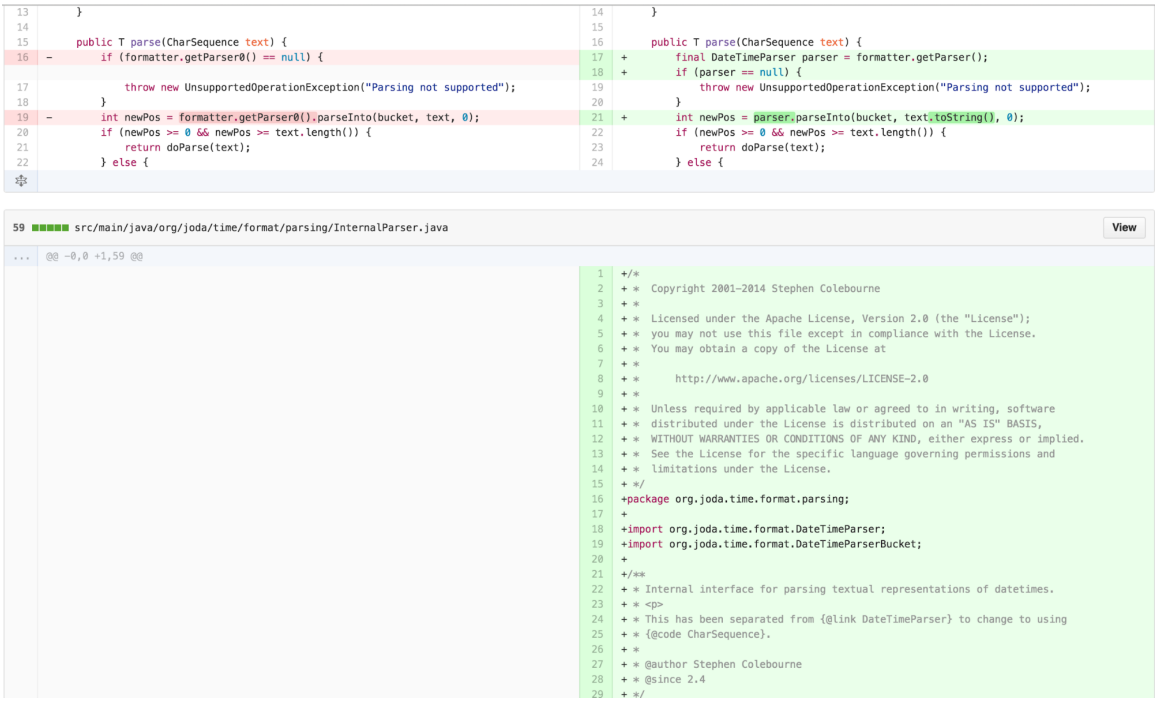


Figure C.118: insert caption

Figure C.119: insert caption

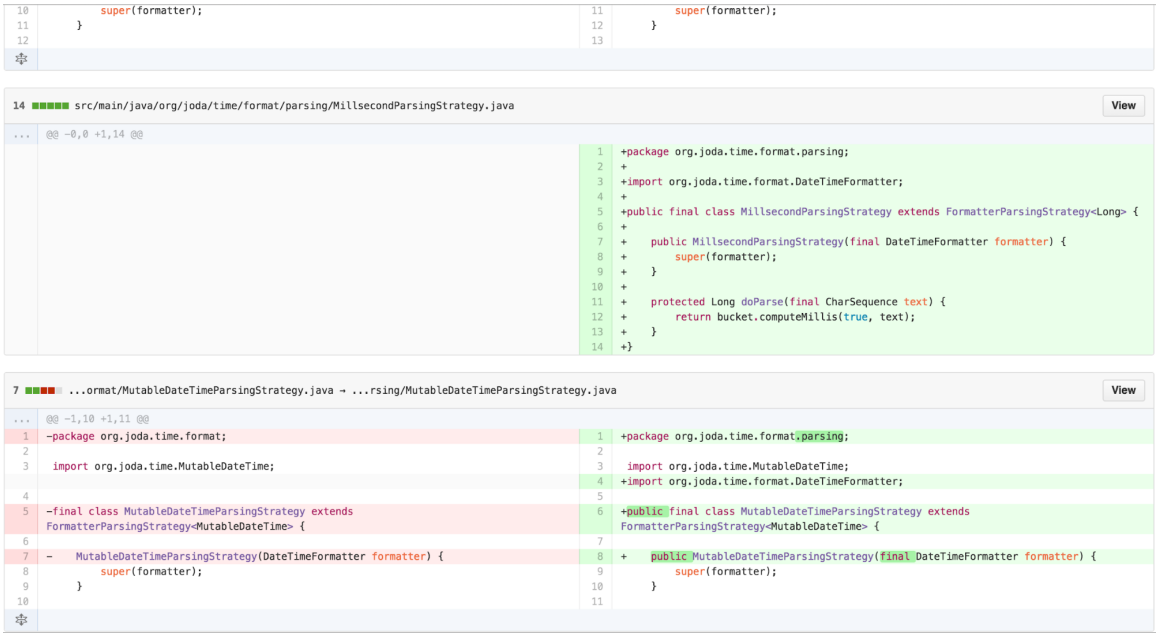


Figure C.120: insert caption

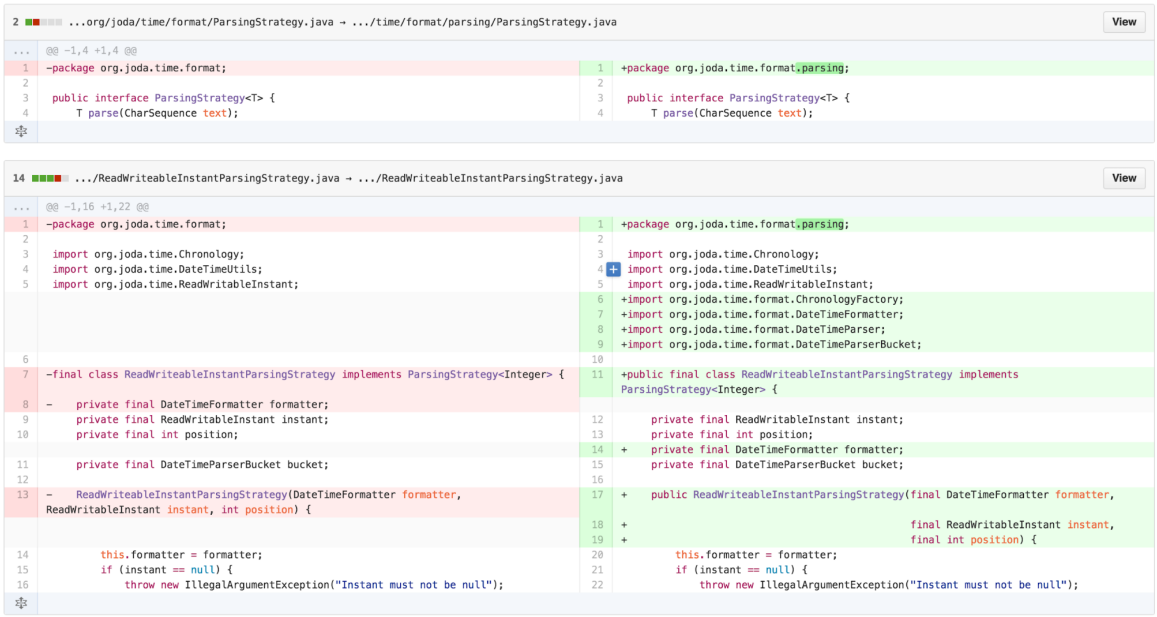


Figure C.121: insert caption

Another heuristic to reduce the number of classes is to find LAZY CLASS and investigate whether we can MOVE METHOD. As `ChronologyFactory` is merely exposing a method that could be a STATIC FACTORY METHOD on `Chronology`, we move the functionality over.

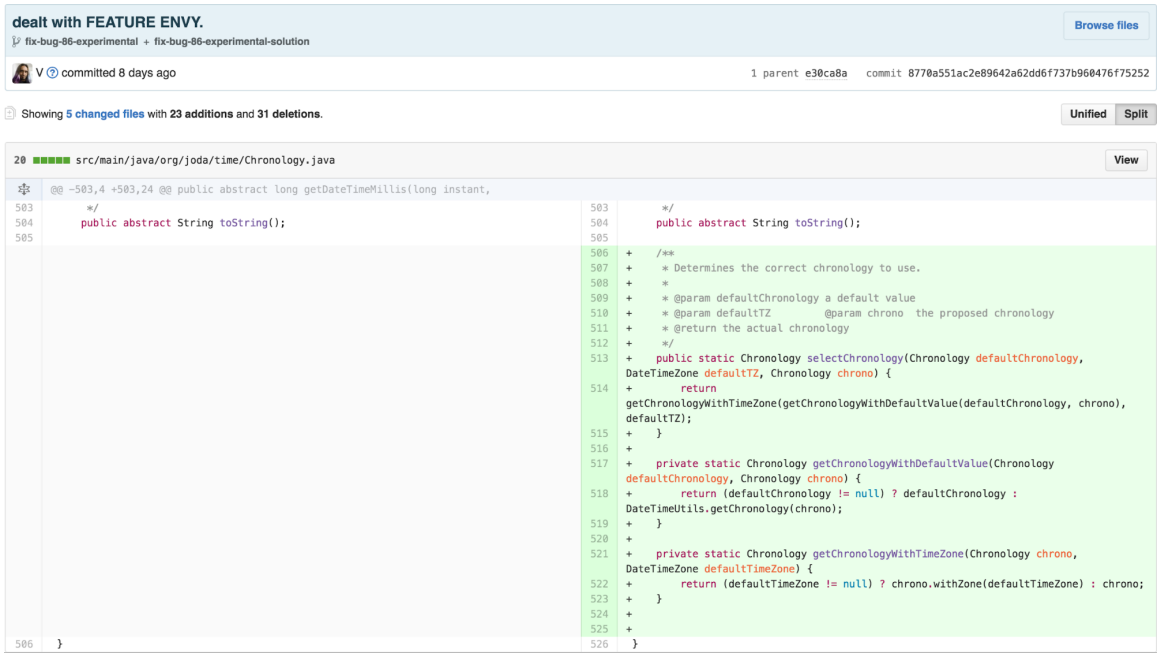


Figure C.122: insert caption

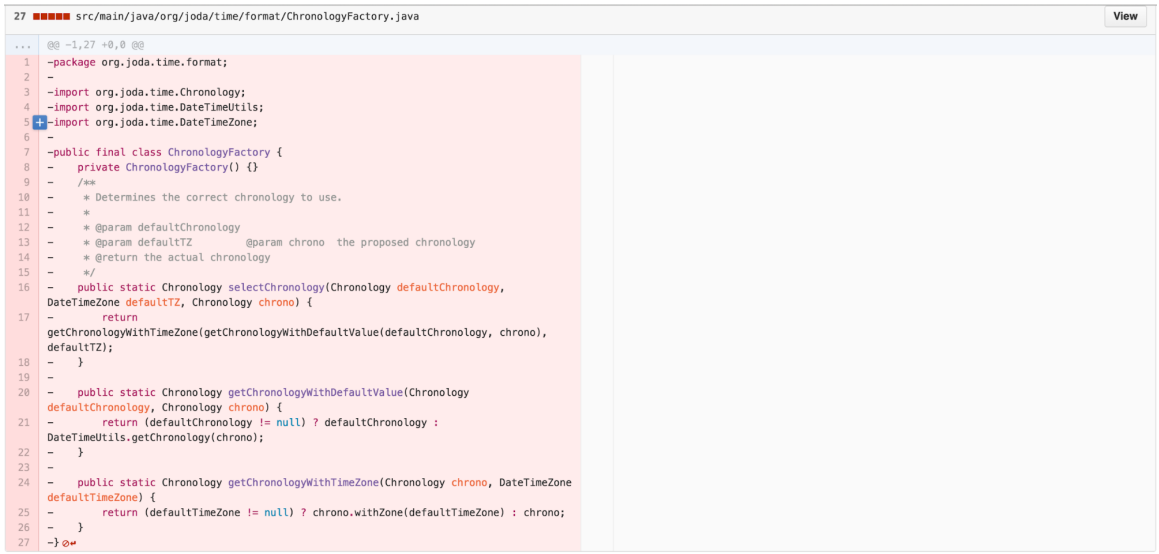


Figure C.123: insert caption

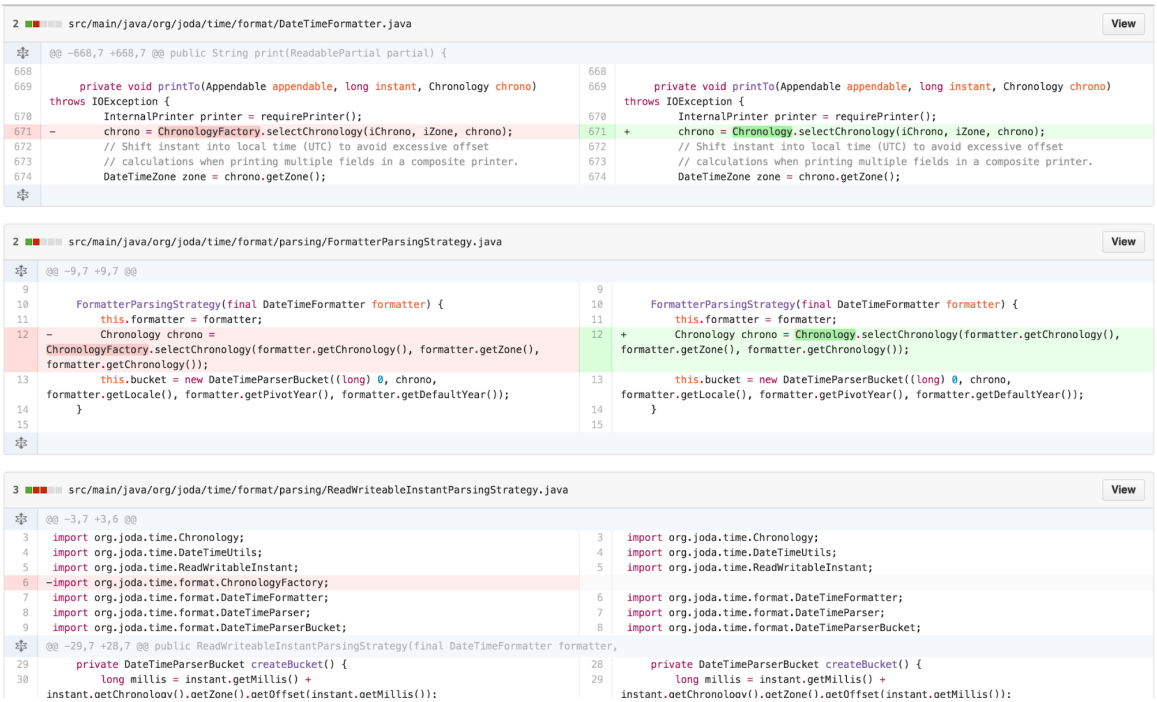


Figure C.124: insert caption

C.6.4 Making Derived Components Expressive, the value of names

Are we done? The resultant code is structurally simpler than before. The outstanding question is, does it communicate the design intent of the original algorithm in a concise and clear way? Without access to the direct authors of the code and given the complexity of the original, it's difficult to say. Debugging through the code flow into the area of the bug, we see much of the code is now delegation: `DateTimeFormatter` ->DateTimeParsingStrategy ->FormatterParsingStrategy ->NumberFormatter ->OffsetCalculator. Most of the work involved in this bug is in the OffsetCalculator...could the code be clearer?

Consultation with other engineers suggested that the OffsetCalculator has variable names that were mostly consequential to its original sources. Revisiting those names may enhance clarity. Additionally, much of the work of calculateLength is done once, ultimately to be used by calcluateValue. This work can be moved to the constructor, with expressive variables assigned. I can clean this up further.

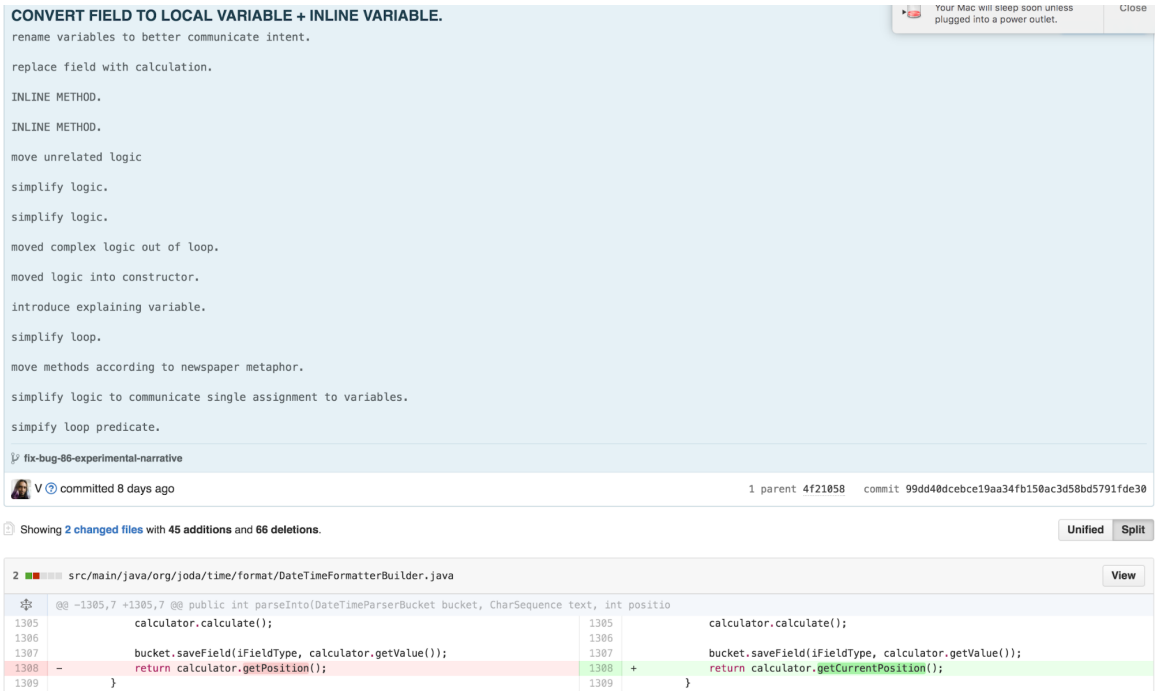


Figure C.125: insert caption

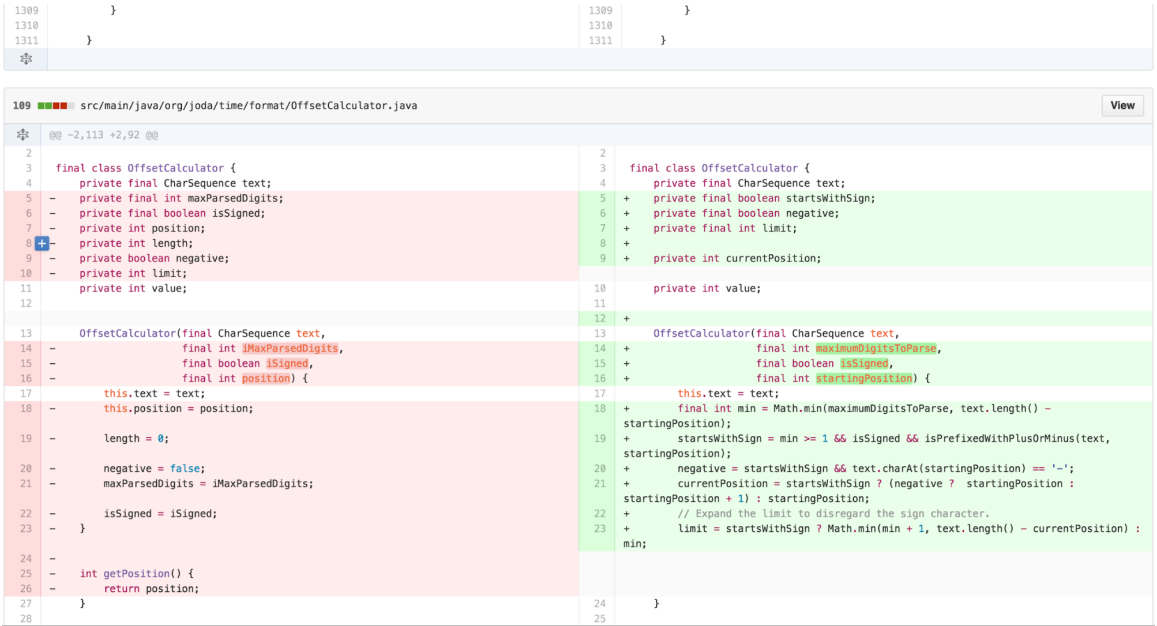


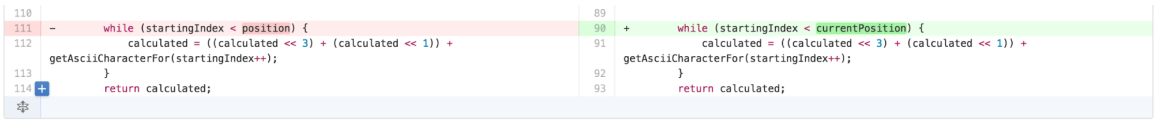
Figure C.126: insert caption

```
-    int getValue() {
-        return value;
     }

-    void calculate() {
-        calculateLength();
-        updatePositionAndValue();
-    }
-
-    private void calculateLength() {
-        limit = Math.min(maxParsedDigits, text.length() - position);
-        while (length < limit && shouldContinue()) {
-            updateBasedOnSign();
-            length = length + 1;
-        }
-    }
-
-    private boolean shouldContinue() {
-        final boolean hasSign = isPrefixedWithPlusOrMinus() && isSigned;
-        return Character.isDigit(text.charAt(position + length)) || hasSign;
     }

-    private boolean isPrefixedWithPlusOrMinus() {
-        final int index = position + length;
-        final char currentCharacter = text.charAt(index);
-        final boolean isFirstCharacterOperator = length == 0 &&
isCharacterOperator(currentCharacter);
-        final boolean hasNextDigitCharacter = index < text.length() - 1 &&
Character.isDigit(text.charAt(index + 1));
-        return isFirstCharacterOperator && isBeforeBoundary() &&
hasNextDigitCharacter;
     }

-    private static boolean isCharacterOperator(final char currentCharacter) {
-        return currentCharacter == '-' || currentCharacter == '+';
     }

-    private boolean isBeforeBoundary() {
-        return length + 1 <= limit;
     }

+    private static boolean isPrefixedWithPlusOrMinus(final CharSequence text, final
int startingPosition) {
+        final boolean isFirstCharacterOperator =
isCharacterOperator(text.charAt(startingPosition));
+        final boolean hasNextDigitCharacter = startingPosition < text.length() - 1 &&
Character.isDigit(text.charAt(startingPosition + 1));
+        return isFirstCharacterOperator && hasNextDigitCharacter;
     }

+    private static boolean isCharacterOperator(final char currentCharacter) {
+        return currentCharacter == '-' || currentCharacter == '+';
     }

+    int getCurrentPosition() {
+        return currentPosition;
     }

+    int getValue() {
+        return value;
     }

+    void calculate() {
+        updatePositionAndValue(calculateLength());
     }
```

Figure C.127: insert caption

```
-    private void updateBasedOnSign() {
-        if (isPrefixedWithPlusOrMinus()) {
-            negative = text.charAt(position + length) == '-';

-            length = negative ? length + 1 : length;
-            position = negative ? position : position + 1;
-            // Expand the limit to disregard the sign character.
-            limit = Math.min(limit + 1, text.length() - position);
         }

     }

-    private void updatePositionAndValue() {
         if (length == 0) {
-            position = ~position;
         } else if (length >= 9) {
-            useDefaultParser();
         } else {
-            useFastParser();
         }
     }

-    private void useDefaultParser() {
         // Since value may exceed integer limits, use stock parser
         // which checks for this.
-        final String toParse = text.subSequence(position, position +
length).toString();
         value = Integer.parseInt(toParse);
-        position += length;
     }

-    private void useFastParser() {
-        int i = negative ? position + 1 : position;

         final int index = i++;
         if (index < text.length()) {
-            position += length;
             value = negative ? -calculateValue(i, index) : calculateValue(i, index);
         } else {
-            position = ~position;
         }
     }

     private int calculateValue(final int i, final int index) {
         int startingIndex = i;
         int calculated = getAsciiCharacterFor(index);

+    private int calculateLength() {
+        int length = startsWithSign ? 1 : 0;
+        while (length + 1 <= limit && Character.isDigit(text.charAt(currentPosition +
length))) {
+            length = length + 1;

         }
+        return length;
     }

+    private void updatePositionAndValue(int length) {
         if (length == 0) {
+            currentPosition = ~currentPosition;
         } else if (length >= 9) {
+            useDefaultParser(length);
         } else {
+            useFastParser(length);
         }
     }

+    private void useDefaultParser(int length) {
         // Since value may exceed integer limits, use stock parser
         // which checks for this.
+        final String toParse = text.subSequence(currentPosition, currentPosition +
length).toString();
         value = Integer.parseInt(toParse);
+        currentPosition += length;
     }

+    private void useFastParser(int length) {
+        int i = negative ? currentPosition + 1 : currentPosition;

         final int index = i++;
         if (index < text.length()) {
+            currentPosition += length;
             value = negative ? -calculateValue(i, index) : calculateValue(i, index);
         } else {
+            currentPosition = ~currentPosition;
         }
     }

     private int calculateValue(final int i, final int index) {
         int startingIndex = i;
         int calculated = getAsciiCharacterFor(index);
```

Figure C.128: insert caption



```
110                                                                                                 89
111 -       while (startingIndex < position) {                                                      90 +        while (startingIndex < currentPosition) {
112             calculated = ((calculated << 3) + (calculated << 1)) +                              91             calculated = ((calculated << 3) + (calculated << 1)) +
    getAsciiCharacterFor(startingIndex++);                                                              getAsciiCharacterFor(startingIndex++);
113         }                                                                                       92         }
114         return calculated;                                                                      93         return calculated;
```

Figure C.129: insert caption

The clearer variable names and re-arranged methods lead me to realize there is a clean separation between two distinct concepts missing in the OffsetCalculator. Much of the work it does results from FEATURE ENVY of the CharSequence. Procedural style code that asks questions such as "does the text begin with a + or -?" would be better answered independently from the calculation.

C.6.5 The Final Form – calculate in the NumberFormatter, NumericSequence

I can decompose the OffsetCalculator back into a data type that provides the necessary functionality on top of CharSequence and a calculation that uses that data type to calculate a value in the NumberFormatter implementation of InternalParser. This aligns most well with the Cognitive Load Theory principle of "write high cohesive content for low-knowledge learners."

EXTRACT PARAMETER OBJECT. Browse files

```
move negative to NumericSequence.

move currentPosition to NumericSequence.

encapsulate CharSequence in NumericSequence.

added Query to see if character at an index is a digit to NumericSequence.

move limit to NumericSequence.

COMMAND QUERY SEPARATION on NumericSequence construction and value calculation.

move getAsciiCharacterAtIndex to NumericSequence.

move length calculation to NumericSequence.

replace low-level details with query.

cleanup.

pull up check.

simplify.:
)

move new NumericSequence class to top-level.

separate command and query.

remove OffsetCalculator and move calculate() back to NumberFormatter.

simplify fields/

moved NumericSequence to parsing package.

move index knowledge to NumericSequence.

cache calculation of length in constructor.

simplify.

Newspaper methaphor.

realign.
```

Figure C.130: insert caption

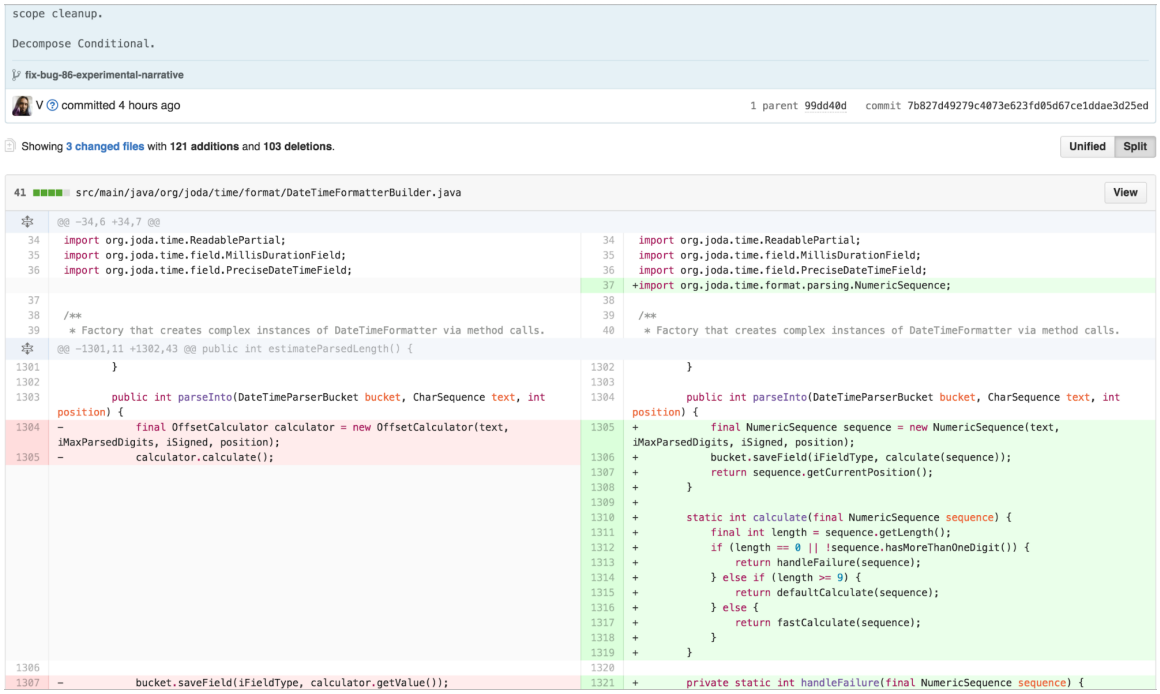


Figure C.131: insert caption

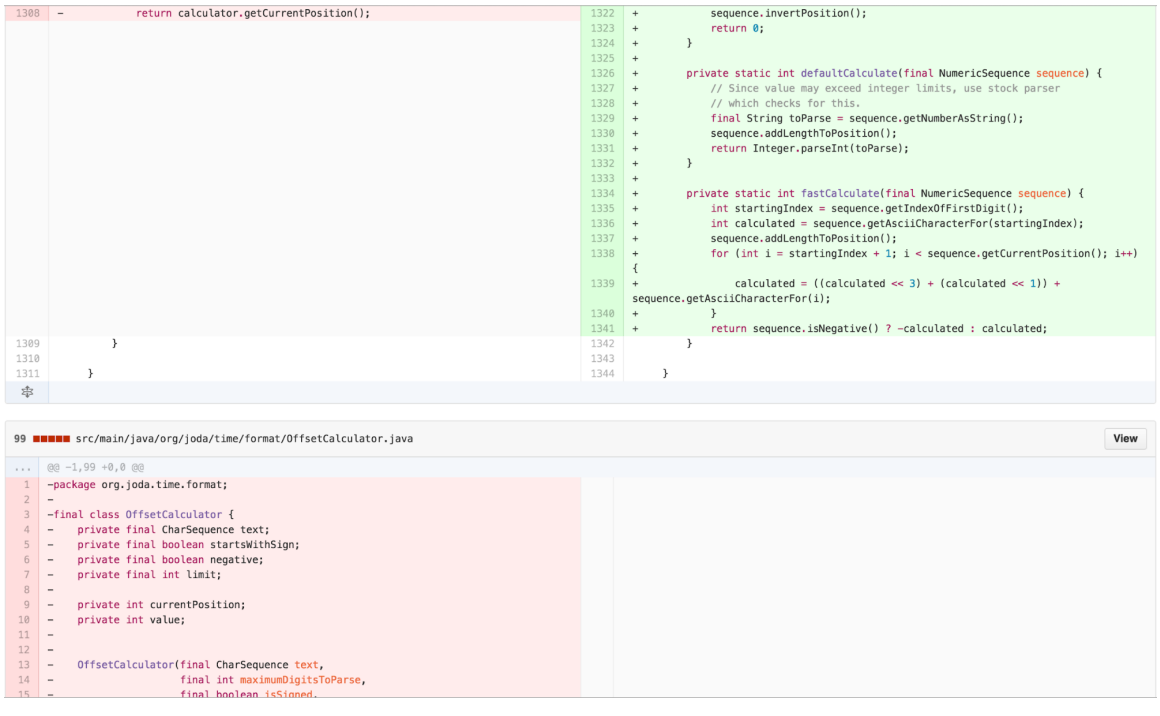


Figure C.132: insert caption

```
-                   final int startingPosition) {
-       this.text = text;
-       final int min = Math.min(maximumDigitsToParse, text.length() -
startingPosition);
-       startsWithSign = min >= 1 && isSigned && isPrefixedWithPlusOrMinus(text,
startingPosition);
-       negative = startsWithSign && text.charAt(startingPosition) == '-';
-       currentPosition = startsWithSign ? (negative ?  startingPosition :
startingPosition + 1) : startingPosition;
-       // Expand the limit to disregard the sign character.
-       limit = startsWithSign ? Math.min(min + 1, text.length() - currentPosition) :
min;
-   }
-
-   private static boolean isPrefixedWithPlusOrMinus(final CharSequence text, final
int startingPosition) {
-       final boolean isFirstCharacterOperator =
isCharacterOperator(text.charAt(startingPosition));
-       final boolean hasNextDigitCharacter = startingPosition < text.length() - 1 &&
Character.isDigit(text.charAt(startingPosition + 1));
-       return isFirstCharacterOperator && hasNextDigitCharacter;
-   }
-
-   private static boolean isCharacterOperator(final char currentCharacter) {
-       return currentCharacter == '-' || currentCharacter == '+';
-   }
-
-   int getCurrentPosition() {
-       return currentPosition;
-   }
-
-   int getValue() {
-       return value;
-   }
-
-   void calculate() {
-       updatePositionAndValue(calculateLength());
-   }
-
-   private int calculateLength() {
-       int length = startsWithSign ? 1 : 0;
-       while (length + 1 <= limit && Character.isDigit(text.charAt(currentPosition +
length))) {
-           length = length + 1;
-       }
-       return length;
```

Figure C.133: insert caption

```
-   }
-
-   private void updatePositionAndValue(int length) {
-       if (length == 0) {
-           currentPosition = ~currentPosition;
-       } else if (length >= 9) {
-           useDefaultParser(length);
-       } else {
-           useFastParser(length);
-       }
-   }
-
-   private void useDefaultParser(int length) {
-       // Since value may exceed integer limits, use stock parser
-       // which checks for this.
-       final String toParse = text.subSequence(currentPosition, currentPosition +
length).toString();
-       value = Integer.parseInt(toParse);
-       currentPosition += length;
-   }
-
-   private void useFastParser(int length) {
-       int i = negative ? currentPosition + 1 : currentPosition;
-
-       final int index = i++;
-       if (index < text.length()) {
-           currentPosition += length;
-           value = negative ? -calculateValue(i, index) : calculateValue(i, index);
-       } else {
-           currentPosition = ~currentPosition;
-       }
-   }
-
-   private int calculateValue(final int i, final int index) {
-       int startingIndex = i;
-       int calculated = getAsciiCharacterFor(index);
-
-       while (startingIndex < currentPosition) {
-           calculated = ((calculated << 3) + (calculated << 1)) +
getAsciiCharacterFor(startingIndex++);
-       }
-       return calculated;
-   }
-
-   private int getAsciiCharacterFor(final int index) {
-       return text.charAt(index) - '0';
```

Figure C.134: insert caption

```
98  -    }
99  -}
```

84 ■■■■■ src/main/java/org/joda/time/format/parsing/NumericSequence.java View

```
...  @@ -0,0 +1,84 @@
1   +package org.joda.time.format.parsing;
2   +
3   +public final class NumericSequence {
4   +    private final CharSequence text;
5   +    private final boolean startsWithSign;
6   +    private final boolean negative;
7   +    private final int limit;
8   +    private final int length;
9   +
10  +    private int currentPosition;
11  +
12  +    public NumericSequence(final CharSequence text, final int maximumDigitsToParse, final boolean isSigned, final int startingPosition) {
13  +        this.text = text;
14  +        startsWithSign = isPrefixedWithPlusOrMinus(maximumDigitsToParse, isSigned, startingPosition);
15  +        negative = startsWithSign && charAt(startingPosition) == '-';
16  +        // Expand the limit to disregard the sign character.
17  +        currentPosition = startsWithSign ? (isNegative() ? startingPosition : startingPosition + 1) : startingPosition;
18  +        limit = Math.min(startsWithSign ? maximumDigitsToParse + 1 : maximumDigitsToParse, text.length() - getCurrentPosition());
19  +        length = calculateLength();
20  +    }
21  +
22  +    private boolean isPrefixedWithPlusOrMinus(final int maximumDigitsToParse, final boolean isSigned, final int startingPosition) {
23  +        final int min = Math.min(maximumDigitsToParse, text.length() - startingPosition);
24  +        return min >= 1 && isSigned && isCharacterOperator(charAt(startingPosition)) && hasNextDigitCharacter(startingPosition);
25  +    }
26  +
27  +    private char charAt(int index) {
28  +        return text.charAt(index);
29  +    }
30  +
31  +    private static boolean isCharacterOperator(final char currentCharacter) {
32  +        return currentCharacter == '-' || currentCharacter == '+';
```

Figure C.135: insert caption

```
33  +    }
34  +
35  +    private boolean hasNextDigitCharacter(int startingPosition) {
36  +        return startingPosition < text.length() - 1 && isDigitAt(startingPosition + 1);
37  +    }
38  +
39  +    private boolean isDigitAt(final int index) {
40  +        return Character.isDigit(charAt(index));
41  +    }
42  +
43  +    private int calculateLength() {
44  +        int length = startsWithSign ? 1 : 0;
45  +        while (length + 1 <= limit && isDigitAt(getCurrentPosition() + length)) {
46  +            length = length + 1;
47  +        }
48  +        return length;
49  +    }
50  +
51  +    public int getIndexOfFirstDigit() {
52  +        return isNegative() ? getCurrentPosition() + 1 : getCurrentPosition();
53  +    }
54  +
55  +    public boolean isNegative() {
56  +        return negative;
57  +    }
58  +
59  +    public int getCurrentPosition() {
60  +        return currentPosition;
61  +    }
62  +
63  +    public void addLengthToPosition() {
64  +        currentPosition = getCurrentPosition() + length;
65  +    }
66  +
67  +    public void invertPosition() {
68  +        currentPosition = ~getCurrentPosition();
69  +    }
70  +
71  +    public String getNumberAsString() {
72  +        return text.subSequence(getCurrentPosition(), getCurrentPosition() + length).toString();
73  +    }
74  +
75  +    public boolean hasMoreThanOneDigit() {
76  +        return (isNegative() ? getCurrentPosition() + 1 : getCurrentPosition()) <
```

Figure C.136: insert caption

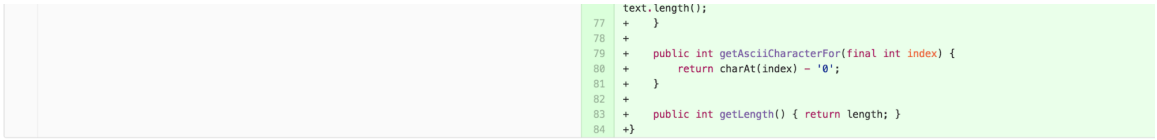


Figure C.137: insert caption

C.7 The Experimental Solution

This has been a lot of work! But the resulting solution is very satisfying. In fact, it requires adding no additional code. The correct solution to fixing the bug actually involves removing some existing confusing code.

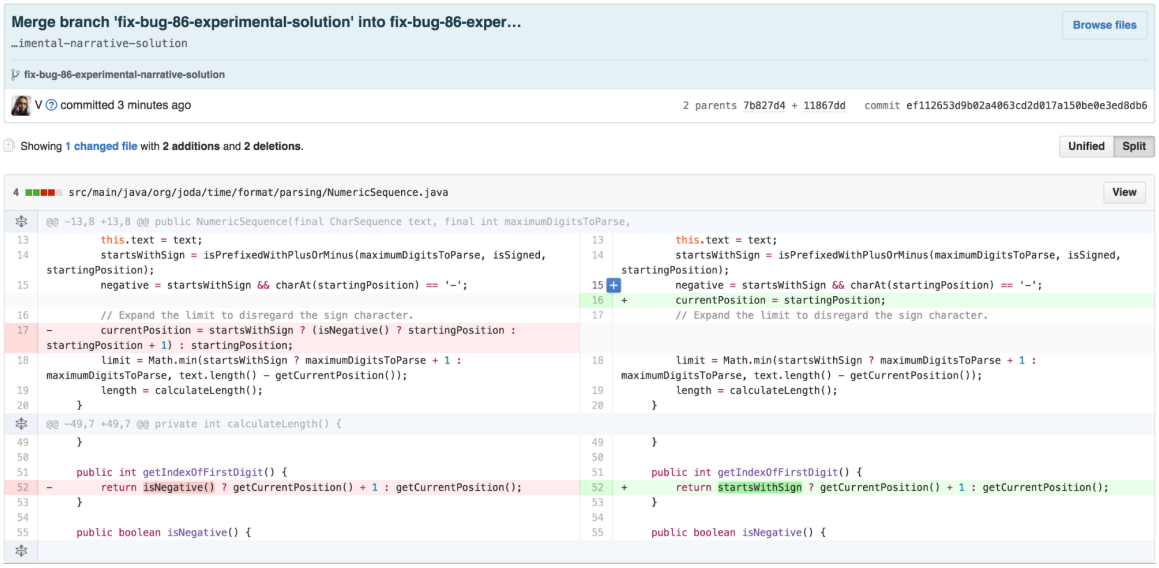


Figure C.138: insert caption

Just like that, the bug is resolved.

C.7.1 Side-by-Side comparison

According to SonarQube 5.4, the metrics for the experimental version are shown below.

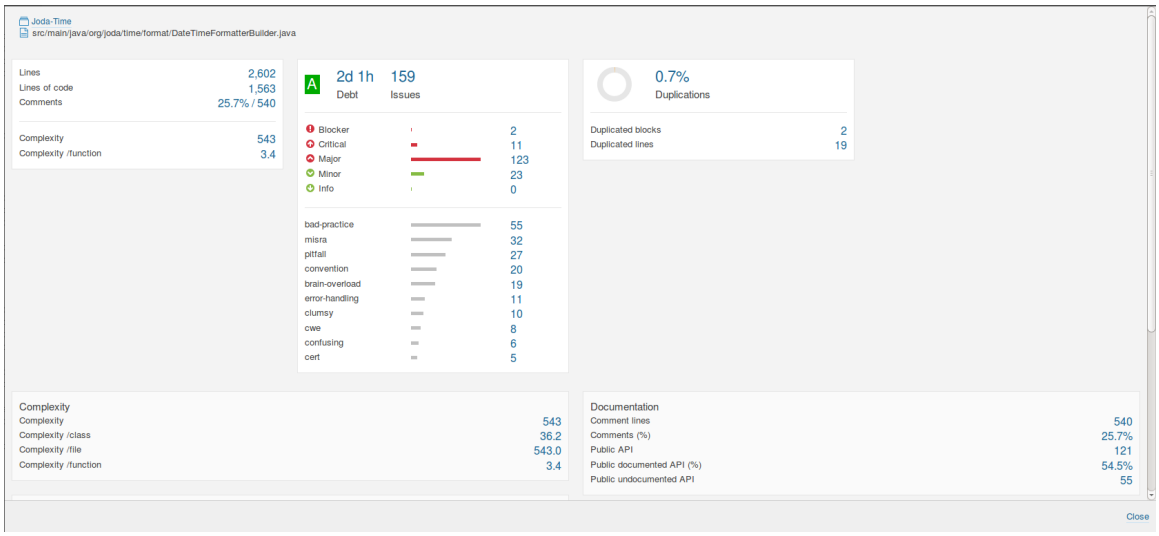


Figure C.139: insert caption

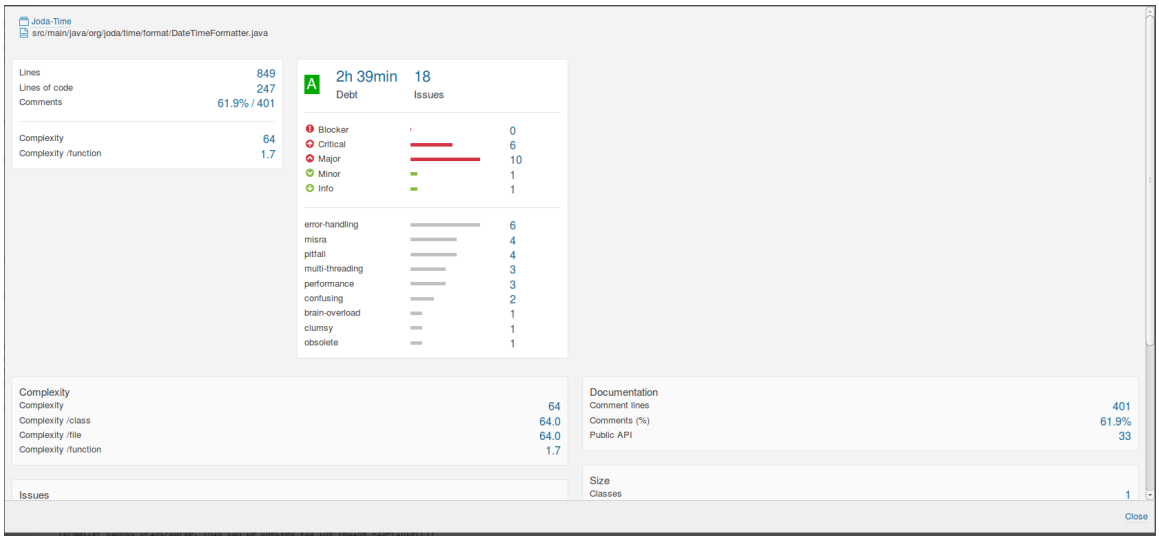


Figure C.140: insert caption

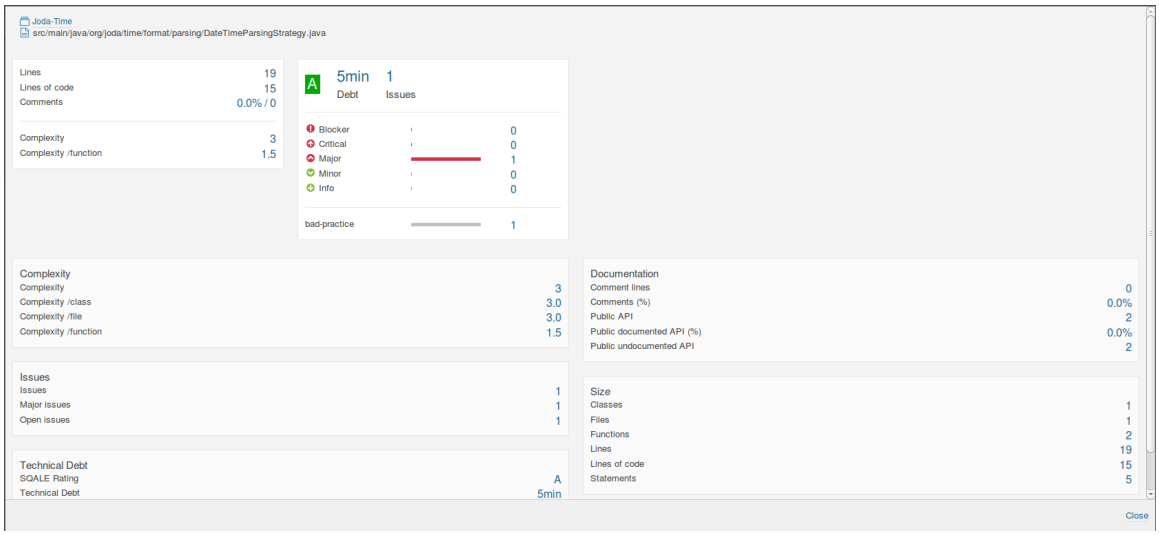


Figure C.141: insert caption

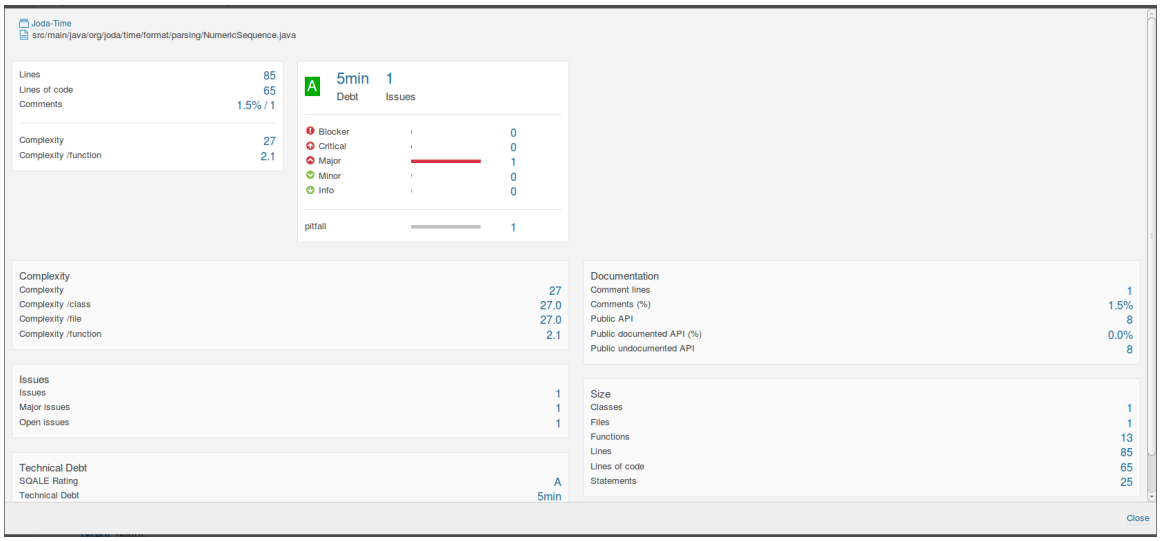


Figure C.142: insert caption

Comparing these with the original DateTimeFormatter and DateTimeFormatterBuilder;

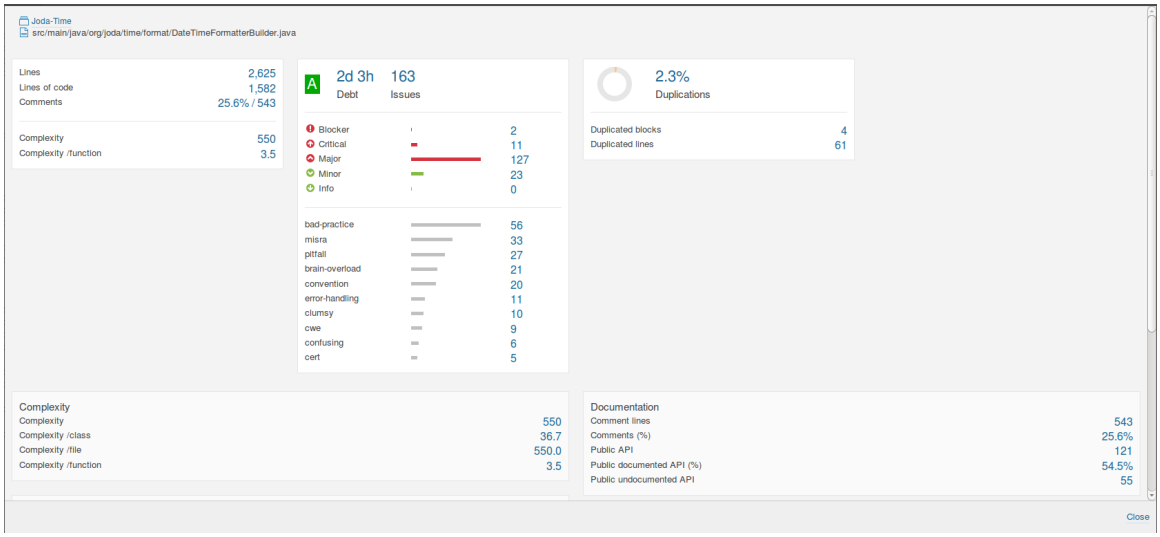


Figure C.143: insert caption

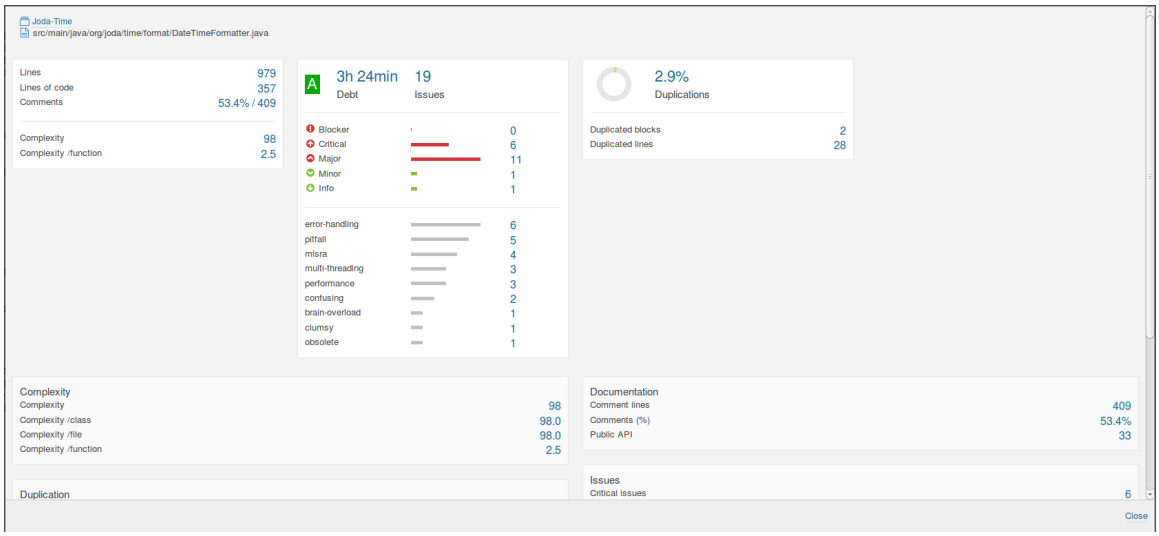


Figure C.144: insert caption

From a pure complexity metrics perspective, it doesn't look like much has changed. DateTimeFormatterBuilder went from a control complexity score of 550 to 543 in the experimental. The complexity per function dropped from 3.5 from to 3.4. The number of lines of code dropped by 21. The number of duplications dropped from 2.3% to 0.7%.

DateTimeFormatter shows a slightly more pronounced effect. The complexity score dropped from 98 to 64. The complexity per function went from 2.5 to 1.7. The number of lines of code dropped by 110. The 2.9% duplications dropped to 0.

Are these changes enough to show a measurable difference in comprehensibility and debugging time? If current best practice software engineering complexity metrics

are sufficient, I do not expect to find a statistically significant difference. Thus, I am ready to conduct a study.