Historic Software Engineering: Insights from Deluxe Paint for the Amiga

Giuseppe Destefanis[a], Yann-Gaël Guéhéneuc[b], Fabio Calefato[c]

[a] *University College London, UK*
[b] *Concordia University, Canada*
[c] *University of Bari, Italy*

Abstract

This article studies and discusses *Deluxe Paint*, a significant graphics program released in 1985 for the Amiga platform. Developed at a time when hardware constraints were severe, *Deluxe Paint* (*DPaint*) was written in C and designed to run on machines with only 256 KB of total memory, divided into video and CPU RAM, and a Motorola 68000 CPU at 7.09 MHz (PAL) or 7.16 MHz (NTSC).

We analyse and study the 17,001 lines of source code of this historic program **to understand the constraints that guided its architecture, design, and implementation** by developers who worked without widespread access to formal programming patterns, reference books, or online resources.

Our analysis covers the overall development, compilation, and quality of *DPaint*. We also discuss the architectural styles, design practices, and implementation idioms present in its source code, as well as its code complexity. We identify antecedents to 11 of the 23 Gang of Four design patterns, all implemented through C constructs, nine years before the GoF catalogue was published, plus seven Amiga platform-specific architectural idioms with no GoF counterpart.

We also show how *DPaint* exploited the compiler and the Amiga hardware for efficiency, including direct graphics manipulation and optimised memory access. These findings highlight the resourceful methods used by its developers to maximise performance on such constrained systems.

Email addresses: `g.destefanis@ucl.ac.uk` (Giuseppe Destefanis), `yann-gael.gueheneuc@concordia.ca` (Yann-Gaël Guéhéneuc), `fabio.calefato@uniba.it` (Fabio Calefato)

By examining the engineering practices of early developers, this study provides insights into software engineering under strict technical limitations. Lessons from *DPaint*, such as memory-conscious design, hardware-aware optimisations, and (un)loadable components, remain relevant today, in particular for IoT systems. By studying historical software, we emphasise the importance of preserving and learning from the past.

Keywords: Deluxe Paint, Amiga, Commodore, Static Analyses, History

1. Introduction

Deluxe Paint [1] (*DPaint*) was developed in 1985 for the Commodore Amiga platform [2] and became a defining graphics software of its era. It is also arguably the "killer app" that popularised the Amiga platform and made it the benchmark for graphics in the late 80s and early 90s.

Designed to run efficiently on hardware with severe constraints, *DPaint* was written in C and initially targeted the Commodore Amiga 1000 computer model. The Amiga 1000, introduced in 1985, featured a Motorola 68000 processor running at 7.09 MHz, with 256 KB of RAM by default, for both video and general-purpose RAM, upgradeable to 512 KB[1], and advanced graphics capabilities for the time, including a 12-bit colour palette and multiple graphic modes, thanks to its dedicated graphics and sound chips.

Despite these hardware limitations, *DPaint* offered advanced graphics capabilities while remaining accessible to a broad user base. It was widely used in digital art and design during the late 1980s and early 1990s. Its use by artists, including Andy Warhol[2], illustrates its application as an early tool for experimentation with digital artistic techniques, used to draw the pixel art of many computer games, e.g., *The Secret of Monkey Island* and *Doom*.

Published by Electronic Arts (EA) [1], *DPaint* emerged shortly after the launch of the Amiga, a platform recognised for its focus on creative computing. Its success resulted from a combination of factors: an intuitive interface, a wide range of features, and the custom chipset of the Amiga. Components such as the Blitter (a co-processor for fast block memory copies and graphics operations) and Copper (a co-processor that synchronises graphics

[1] Memory could be expanded to 8.5 MB with expensive external extensions that were available later in the life of the computer model.

[2] `https://www.warhol.org/exhibition/warhol-and-the-amiga/`

Figure 1: Screenshot of *DPaint* after starting the program

changes with the video beam) enabled efficient graphics operations, supporting real-time image manipulation. The adoption of the Interchange File Format (IFF), developed by Jerry Morrison at EA, further contributed to its impact by allowing interoperability between Amiga (and non-Amiga) applications. Figure 1 shows the user interface of *DPaint* in its first version.

The design of *DPaint* was influenced by earlier graphics software, such as *MacPaint* and *SuperPaint*, but introduced a broader set of tools, suited for professional use. Its interface was designed to be straightforward while offering powerful features, attracting both hobbyists and professionals. Over multiple versions (I through V), *DPaint* introduced features that expanded the possibilities of digital art. Unique features of *DPaint* included support for the Extra-Half-Brite (EHB) graphics mode, enabling 32 base colours and 32 half-bright variations; the Hold-and-Modify (HAM) graphics mode, which expanded the colour range to 4,096; and, on the later Commodore Amiga Advanced Graphics Architecture (AGA) graphics chipsets, the support of 256 colours from a 24-bit palette. They also included animation support, starting in *Deluxe Paint III*, to create multi-frame animations.

Although widely adopted, *DPaint* was ultimately tied to the fate of the Amiga platform. As Commodore faced financial difficulties in the mid-1990s, the Amiga market declined, leading to its discontinuation. However, its legacy persists in later graphics software, including Adobe Photoshop and

GIMP, which adopted some of its features, such as structured layers, blending modes, and pixel editing. It is also still used in retro computing communities, particularly for pixel art creation and retro-game development.

In 2015, EA released the source code of *Deluxe Paint I*, through the Computer History Museum of Mountain View, CA, USA. This source code, written in C, consists of 88 source files (77 `.C` and 11 `.H`; 89 including the linker control file `PRISM.txt`), containing 17,001 lines of code and 546 functions. We obtained its source code from the Computer History Museum[3], which we studied with the following goal:

We want to understand the constraints that guided the architecture, design, and implementation of *Deluxe Paint* to handle the computational and memory constraints of the Amiga 1000.

Our goal is motivated by the following three observations. First, the software was developed in 1985, predating the Gang of Four's *Design Patterns* book (1994), the widespread adoption of object-oriented programming, and the existence of online developer communities and the Internet itself. Developers had no access to Stack Overflow, no UML for architectural modelling, and no formal software engineering methodologies. Second, despite these constraints, *DPaint* achieved commercial success and remained in active development for nearly a decade, suggesting that its developers employed effective engineering practices. Third, the hardware constraints of the Commodore Amiga 1000 required optimisation strategies that differ substantially from those necessary for software development for modern platforms, but that are similar to those for development for IoT devices.

To reach our goal, we answer five research questions:

RQ1: How and by whom was *DPaint* developed, and what was its overall quality?

We examine the source code qualitatively, studying its file organisation, naming conventions, and contributor attributions extracted from code comments. We report on the compilation infrastructure described in the linker control file and recompile the source with a modern SAS/C compiler to assess code completeness and quality. Quantitatively, we compute lines of code, function counts, and cyclomatic complexity at both file and function levels.

[3] `https://computerhistory.org/blog/electronic-arts-deluxepaint-early-source-code/`

RQ2: What architectural styles can be observed in the source code of *DPaint*?

We construct a directed graph from the 465 file-to-file dependencies and apply graph-theoretic centrality measures (betweenness, in-degree, eigenvector, PageRank) to identify structurally significant files. We use community detection to reveal natural file groupings and robustness analysis to assess architectural resilience to the removal of hub files.

RQ3: What design patterns are present in the source code of *DPaint*?

In 1985, developers had no access to design patterns literature, object-oriented programming standards, or formal architectural guidelines. We examine if recognisable design patterns emerged organically from the problem domain and hardware constraints through a systematic manual inspection of the source code, identifying structural, behavioural, and creational constructs that correspond to patterns later formalised by Gamma et al.

RQ4: What coding idioms are present in the source code of *DPaint*?

The Amiga platform divides memory into Chip RAM (accessible by both custom graphics chips and the CPU) and Fast RAM (CPU-only), requiring explicit management decisions at the application level. We identify coding idioms that have no counterpart in the GoF catalogue but reflect deliberate engineering responses to these hardware constraints, including a memory overlay system (i.e., loading code segments on demand to fit limited RAM), cooperative resource sharing between the application and the operating system, and direct hardware register access patterns.

(Throughout this article, we use patterns for solutions that have Gang-of-Four analogues, and idioms for platform-specific solutions with no GoF counterpart.)

RQ5: How was complexity distributed across the different components of *DPaint*, and why? Graphics software involves algorithmically complex operations: image transformations, colour palette manipulation, real-time rendering, and direct hardware register access. We analyse cyclomatic complexity of both functions and files to determine if certain types of functionality (e.g., graphics algorithms vs. user interface code) consistently exhibit higher complexity. We also examine the relationships between complexity, code size, nesting depth, and documentation density. Understanding these distributions provides insight into where developers invested effort and how they managed complexity without modern abstractions.

Through this analysis, we offer six contributions:

- A qualitative and quantitative characterisation of *DPaint* development context, contributors, and code quality, including its recompilation with a modern compiler.
- A dependency network analysis revealing the architectural structure of a 1985 graphics application, including identification of hub files and module boundaries.
- An identification of antecedents to 11 of the 23 GoF design patterns and 7 platform-specific idioms in imperative, non-object-oriented, pre-ANSI C code that predates the GoF catalogue by nine years.
- A complexity distribution analysis across 546 functions and 88 files, categorised by functional purpose.
- Documentation of memory optimisation and hardware-aware programming techniques employed under extreme resource constraints.
- A reproducible analysis methodology, including a complete replication package with data, code, patch, etc.

We use this understanding to draw lessons and parallels with modern software development. We conclude that the techniques used 40 years ago still apply today, and their "rediscovery" could benefit current software development, especially for hardware-constrained IoT devices.

Outline. Section 2 reviews related work on the historical study of hardware, video games, and software artefacts. Section 3 describes our method for each research question. Section 4 presents the results. Section 5 draws lessons and discusses threats to validity. Section 6 concludes with future work.

2. Related Work

This article relates to the many works on the history of computer-related artefacts, which we can divide into three main topics: computer hardware, video games, and non-game software artefacts. Although our study focuses on non-game software artefacts, we review related work in other topics to justify its importance further.

2.1. Historical Study of Computer Hardware

The historical study of (old) computer hardware itself divides into three main topics: nostalgia, preservation, and education.

One of the main reasons for individuals to seek, maintain, and use old computer hardware is *nostalgia*. Nostalgia encompasses both memories of the past and the desire to relive this past, even if this past really never existed, at least not as remembered. For example, Alizadeh et al. [3] reported that participants in a study on nostalgia for technology saw "older technologies with positive connotations and shared memories of how they had adapted and appropriated these technologies, rather than normative uses". Ismail et al. [4] showed the benefit of such nostalgia with a "consistent and robust evidence for the positive effects of nostalgic reminiscence". Thus, through nostalgia, the historical study of computer hardware can have a positive impact on the mental health of people performing it.

Another reason for individuals and organisations to study computer hardware from a historical perspective is *preservation*. Although migration and emulation are seen "as the two ends of a continuum of preservation" by many an archivist, Galloway [5] explained that "an unavoidable challenge lies at the beginning of the chain of digital preservation: the bits have to be captured into a current preservation space, and to recover them we have to be able to read them and copy them, which cannot be done without old hardware". She reported that preservation is often done by non- and semi-professionals, with expertise in computer software and hardware. Despite the lack of any formal museological education, they bring deep technological and historical expertise and, often, first-hand experience.

Yet another reason for studying old computer hardware is *education*. Indeed, old computers are simpler than their current (and presumably future) counterparts, but they also include many clever techniques that allowed computer and software engineers to go further than was possible at the time. For example, Tomari and Hiraki [6] used 270 old computers from between 1979 and 2014 to teach students different computer architectures and the contexts and reasons for today's standard features. They also reported the power consumption and computation benchmarks of various architectures, including ARM, m68k, PowerPC, and x86 and x86_64. Similarly, Zhang [7] proposed "an initiative to introduce retrocomputing to bridge the gap between hardware and software curricula" and that between "the simplicity of theory and complicated real-world practical systems". Generally, they concluded that "retrocomputing as an integrative STEM education program brings several

long-term benefits" such as motivation, learning habits, and awareness of sustainability problems.

At the intersection of hardware and video games lies *archaeogaming* [8], which sits at the "intersection of archaeology and video games and applied archaeological methods and theory to understanding game-spaces as both site and artifact". Archaeogaming considers video games in their entirety, from their physical manifestation and location in (real) space and time, to their implementation (related to Aycock's work below in subsection 2.2), to the spaces and artefacts *inside* computer games. It adapts and applies the methods of archaeology to video games, including their hardware and the hardware on which they ran. In his book [8], Reinhard dedicates the first chapter to the excavation of the so-called "Atari video game burial", located in a landfill site in Alamogordo, NM, USA, that uncovered unsold cartridges of E.T. the Extra-Terrestrial among other physical artefacts. This excavation followed the methods and standards of archaeology.

Summary and Relation to Our Study. Although we are not studying directly any hardware in this article, we cannot deny that our interest in *DPaint* stems from our curiosity about how it uses the specialised chips of the Amiga platform. Moreover, there is also some nostalgia in studying *DPaint*, which the authors used on their favourite computer platform of the time.

2.2. Historical Study of Video Games

Many articles and books exist on the history of computer games. While many books serve as compendia of games, for a particular platform, e.g., the Sinclair ZX Spectrum [9] or the Commodore Amiga platforms [10], or for a particular genre, e.g., computer role-playing games (CRPGs) [11], other books propose methods and studies of games as archaeological artefacts.

In particular, Reinhard [8] defines the concept of digital excavation, which is the use of archaeological methods in and to digital spaces. He illustrates a subset of this concept, that of landscape archaeology, with the game No Man's Sky by Hello Games. He argues that the landscapes in a game like No Man's Sky satisfy the three main topics of landscape archaeology as defined by David and Thomas [12], who are professors and professional archaeologists: (1) fields of human engagement; (2) physical[4] environmental contexts; and (3) representations. The archaeology of digital spaces can inform us

[4]As in having a topography, vegetations, landmarks, etc.

about the agency afforded to players, but also about the ideologies of its developers. In a contribution to Reinhard's book series on Digital Archaeology: Documenting the Anthropocene, Aycock [13] detailed "the procedure and thought process required to reverse engineer a digital artifact", drawing a parallel with archaeological fieldwork, and used Electronic Arts' game Amnesia to exemplify these procedures and processes.

Some studies are interested in the history *of* video games. Using a dataset of 1,725 video games released between 1981 and 2015, Rochat [14] applies a "distant reading approach" and uncovers tendencies at the macro levels in the distribution of the games across platforms, genres, and periods. He reported three periods of growth: a steady growth from 1981 to 1997, an accelerated growth from 1999 to 2007, and a slowed growth from 2008 onwards, possibly due to the ballooning of mobile gaming. More generally, Lawler and Smith [15] argued that social and cultural understandings are embedded in games, which reflect the societies in which they were created. Thus, they and their history should be studied to "reveal new connections and deepen our understanding of space, race, gender, and class".

Meanwhile, other studies focused on the history presented *in* video games. For example, in his book, Rollinger [16] proposed to "[explore] the varied depictions of the ancient world in video games". He argued that classical studies have studied the depiction of classical antiquity in other media, such as films, but not in video games, despite its uses in popular games such as Ubisoft's Assassin's Creed and MicroProse's Civilization. In her review of the book, Ager [17] reports that there is still uncertainty whether historical studies are from the perspective of developers or players and for an audience of players or scholars. Belyaev and Belyaeva [18] pointed out that the history in video games can be used to distort, mythologise, and politicise history, for example, by using unsubstantiated anti-Soviet rhetoric in game plots.

Yet other studies, like that of Spring [19], concern the use of video games *for* presenting historical research and "transforming readers, learners, and viewers into players interacting with history". Spring described how historical research could be presented as a scholarly game, arguing that historical research has inherent game-like qualities while games are popular and can be educational. However, she also warned that "development and iterative research" are still necessary to identify the best practices in translating historical research into gameplay.

Some authors focused on analyses of the software making up historical video games. Aycock [20] described a method and the analyses of over 100

games. He focused on the implementations of these games, in particular the tricks used by developers to work within the constraints of early computers and overcome some of their limitations. He divided his book into seven topics that correspond to core features of many historical video games and presented developers with challenges or features used by developers to overcome these challenges. These topics are (1) memory management, (2) I/O, (3) interpreters, (4) data compression, (5) procedural content generation, (6) protection, and (7) obfuscation and optimisation. For example, developers would design and implement interpreters to simplify the creation of game levels and reduce their physical sizes. For another example, they would put in place procedures to generate game content to reduce game size, improve replayability, and increase game lifespan, as well as the ratio time played/-money spent.

Finally, other studies concern the preservation of video games in the face of two impermanences: that of the physical media on which games are stored and that of the Internet resources that many now require to function. Dym et al. [21] enumerates the problems of preserving and studying games: physical media, Internet servers, and legal challenges. They state that "[a]rchiving video games and consoles requires restoration work that is relatively new compared with preservation practices in mediums such as film or music" [22]. They present the results of 15 semi-structured interviews and divide the participants' answers into four objectives: playing games, modifying games, transforming games, and archival work. They report that lack of access is often a motivation for archiving games and that game companies are seen as gatekeepers at best, opponents at worst, and that advocacy and resources are required to sustain preservation efforts.

Relation to Our Study. Although we do not study a video game in this article, we certainly follow in the footsteps of previous authors who emphasise the value of such study and proposed methods to perform such studies. We thus aim to contribute to the historical study of software artefacts.

2.3. Historical Study of Non-game Software Artefacts

Some studies pertain to the history of particular non-game software artefacts, like programming languages. The ACM SIGPLAN History of Programming Languages conference (HOPL) is dedicated to the "historical treatment of the development of programming languages as a means of human expression and creation." [23]. Although quite infrequent (1978, 1993, 2007, and

2021), the conference features long, in-depth accounts of the history of programming languages or related tools. For example, Monnier and Sperber [24] studied and reported on the history and evolution of Emacs Lisp. They reported that the choice of Lisp was first due to Stallman's context and experience at the time, and second due to its characteristics for a programmable editor. They also showed that its evolution was constrained by backward compatibility but also by advances in language design and changes in maintainership. Other articles in the HOPL conference series include histories of C, C++, Erlang, Smalltalk, and many others.

This article also relates to the many studies on the comprehension and evolution of software systems published in software-engineering conferences, in particular the International Conference on Program Comprehension (ICPC) and the International Conference on Software Maintenance and Evolution (ICSME) conference series. These works propose novel methods to analyse software systems and their evolution, possibly applied to contemporary systems for their evaluation. They are not *specifically* interested in *history* of the analysed systems and on the design and implementation *choices* made in relation to the hardware *constraints* on which they run. Meanwhile, in this article, we analyse a historical software, *DPaint*, to understand, report, and discuss its history and the constraints that guided its architecture, design, and implementation.

Relation to Our Study. Some works studied the history of non-game software artefacts, in particular programming languages. Many others proposed novel methods to study software artefacts with the goals of helping developers and maintainers. In this article, we study a software system to discuss its design and implementation particularities within its historical context.

2.4. Summary

Previous work shows that, beyond nostalgia and museology, the historical study of hardware and software artefacts has several benefits for engineering education and the engineering of small devices, e.g., IoT devices. Students can enjoy the simplicity of past hardware and software and translate lessons learned to current devices, which require clever techniques to make the best use of low, constrained resources.

3. Method

We structure the analysis around five research questions, each employing methods suited to the available data. Figure 2 illustrates its workflow.

Throughout this article, we use the term *file* to refer to an individual `.C` or `.H` source file. We use the term *module* informally to refer to a group of source files that share a naming prefix (e.g., MAG for `MAGBITS.C`, `MAGOPS.C`, `MAGWIN.C`) and exhibit strong internal coupling.

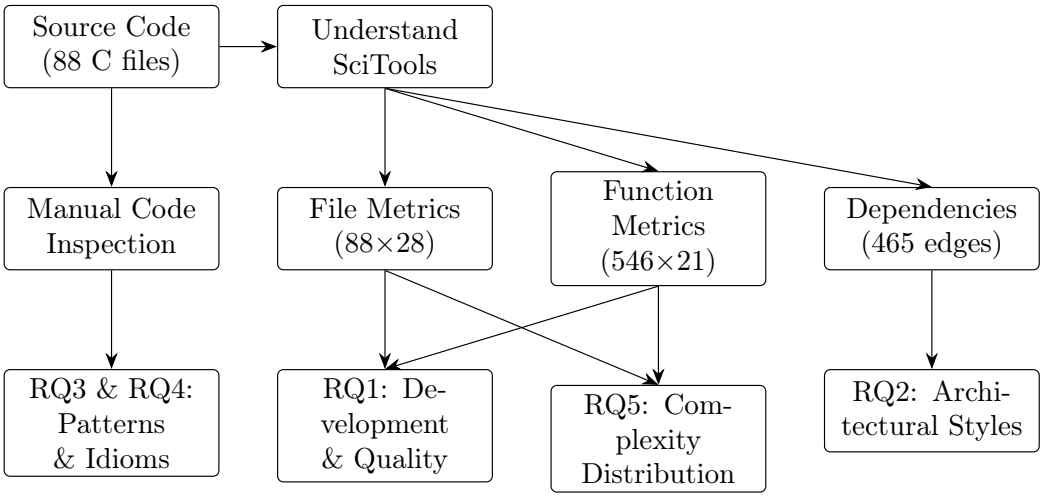

Figure 2: Analysis workflow. Understand SciTools extracts three metric datasets from the 88 source files. File-level and function-level metrics feed RQ1 and RQ5, the dependency graph feeds RQ2, and manual code inspection addresses RQ3 and RQ4.

We organise the method by research question: each subsection below describes the analysis approach for one RQ, so that readers can follow the method and its corresponding results as a self-contained unit.

3.1. RQ1: How and by whom was DPaint developed, and what was its overall quality?

We qualitatively analyse the source code of *DPaint* to report on its files and their naming conventions and contents. In terms of content, we report the number of files, lines of code, structures, and functions. Finally, we study the compilation of *DPaint* and recompile it to discuss coding styles and constraints.

To enable analysis by functional purpose, we categorised files based on filename patterns: colour management (`PALETTE`, `COLOR`, `HSV`), drawing tools

(PAINT, DRAW, BRUSH), file I/O (DPIO, FILE, LOAD, SAVE), view and magnification (MAG, ZOOM, VIEW), user interface (MENU, WINDOW, PANE), graphics core (BLIT, BMOB, GRAPH), transformations (ROTATE, SHEAR, STRETCH, BEND), and system initialisation (INIT, PRISM, SYSTEM).

3.2. RQ2: What architectural styles can be observed in the source code of DPaint?

To characterise the architectural styles of *DPaint*, we performed a network analysis of the file-to-file dependencies extracted by Understand SciTools.[5] This analysis combines graph-theoretic centrality measures, community detection, and robustness simulation to reveal the structural organisation of the source code.

3.2.1. Graph Construction

We constructed a directed graph from all 465 file-to-file dependencies extracted by Understand SciTools using NetworkX [25]. Each edge carries a weight representing dependency strength (number of references). The dependency data includes both internal project files and external Amiga OS headers (e.g., `exec/types.h`, `libraries/dos.h`). Our analysis operates at two levels. The primary level treats each source file as a node and each dependency as a directed edge. A secondary grouping by filename prefix (e.g., MAG for MAGBITS.C, MAGOPS.C, MAGWIN.C) allows us to examine module-level coupling patterns. For module-level analyses, we filtered external dependencies by excluding targets that contain path separators, retaining only internal project relationships.

3.2.2. Centrality Measures

We computed multiple centrality measures to identify structurally important files. Betweenness centrality quantifies how often a node lies on the shortest paths between other nodes [26], identifying bridge files that connect separate parts of the system. In-degree identifies foundational files upon which many files depend. We also computed eigenvector centrality (files connected to important files) and PageRank as alternative importance measures.

For visualisation, we computed node positions using the Fruchterman-Reingold force-directed algorithm [27] with parameters $k = 1.5$, 100 iterations, seed = 42, and edge weights influencing spring strength.

[5] `https://scitools.com`

3.2.3. Community Detection

To identify coherent file groupings—which we call*modules* in the following (i.e., sets of source files sharing a common naming prefix, such as MAG for `MAGBITS.C`, `MAGOPS.C`, and `MAGWIN.C`)—we applied the greedy modularity optimization algorithm [28] on the undirected projection of the dependency graph. This algorithm partitions nodes to maximise modularity without requiring prior knowledge of the grouping structure. We also extracted groupings from file names (uppercase prefixes such as "PAL" from `PALETTE.C`) to examine whether naming choices reflected coupling patterns. This comparison tests whether the developers' naming conventions reflect actual coupling relationships, i.e., whether files sharing a prefix are more strongly coupled to each other than to unrelated files.

3.2.4. Robustness Analysis

To assess architectural resilience, we simulated progressive removal of high-centrality files following the methodology of Albert et al. [29]. We compared two removal strategies: betweenness centrality (targeting architectural bridges) vs. in-degree (targeting foundational files). After each removal, we tracked the number of connected components and the fraction of nodes in the largest component. We removed up to 10 nodes under each strategy.

3.3. RQ3: What design patterns are present in the source code of DPaint?

To identify design patterns in the source code, we conducted a systematic manual inspection of all 77 `.C` source files and 11 header files. For each file, we examined function signatures, struct definitions, macro definitions, dispatch tables, and function pointer usage to identify structural, behavioural, and creational constructs that correspond to patterns later formalised by Gamma et al. [30]. Every pattern identification is grounded in specific file names, function signatures, struct definitions, and line numbers. We distinguish between patterns that arise from deliberate abstraction and those that emerge from hardware necessity, and we note where a pattern is partial relative to its GoF specification. To ensure reproducibility, we developed a verification tool (`verify_patterns.py`) that mechanically confirms the presence of each cited code structure in the source files.

3.4. RQ4: What coding idioms are present in the source code of DPaint?

To characterise the coding idioms of *DPaint*, we combined quantitative source-level metrics with a manual inspection of platform-specific idioms.

The metrics, described first, characterise the coding practices in the source code. The manual inspection, described last, identifies the low-level idioms that have no counterpart in the GoF catalogue and that respond to the hardware constraints of the Amiga 1000.

3.4.1. Code Density and Documentation

We computed code density (executable lines divided by total lines) and comment density (comment lines divided by total lines). High code density with low comment density may reflect the simplicity of well-written code that requires little explanation, the conventions of a single-developer project where the author is the primary reader, or a combination of both. Since comments are stripped at compile time, their absence does not affect binary size; the low comment density is therefore a source-level characteristic, not an optimisation choice.

3.4.2. Function Signature and Preprocessor Analysis

We examined function parameter counts as an indicator of design intent. Lower parameter counts reduce stack usage and may indicate reliance on global state, while higher counts may reflect a deliberate reuse strategy: the same function serves multiple callers by varying its arguments, avoiding code duplication at the cost of a larger stack frame. We also measured preprocessor directive frequency, as heavy macro usage may indicate inline expansion to avoid function call overhead. The Lattice C compiler used for *DPaint* offered no automatic function inlining. Preprocessor macros were therefore the only mechanism for inline expansion, which explains the high macro density observed in performance-sensitive files.

3.4.3. Coupling and Stability Metrics

Using the dependency graph, we computed coupling metrics as defined by Martin [31]. Afferent coupling (Ca) counts incoming dependencies (files that depend on this file); efferent coupling (Ce) counts outgoing dependencies (files on which this file depends). The instability index $I = Ce/(Ce + Ca)$ ranges from 0 (maximally stable, depended upon by others but depending on nothing) to 1 (maximally unstable, depending on others but not depended upon). Files with low instability serve as foundations; files with high instability are more volatile.

3.4.4. Manual Inspection of Platform Idioms

To identify coding idioms specific to the Amiga platform, we examined low-level programming practices that have no direct counterpart in the GoF catalogue but reflect deliberate engineering decisions driven by the hardware constraints of the Amiga 1000. We inspected the 77 `.C` source files and 11 header files for memory management strategies, hardware register access patterns, code overlay techniques, and cooperative resource sharing between applications and the operating system. As in our pattern analysis (Section 3.3), each identified idiom is grounded in specific file names, function signatures, struct definitions, and line numbers.

3.5. RQ5: How was complexity distributed across the different components of DPaint, and why?

To examine how complexity is distributed across functional components, we analysed cyclomatic complexity values at both file and function levels.

3.5.1. Complexity Extraction and Categorisation

Understand SciTools computed cyclomatic complexity for each of the 546 functions using McCabe's definition [32]. For file-level analysis, we computed aggregate metrics: the sum of the complexity of all the functions, the maximum single-function complexity, and the mean function complexity.

3.5.2. Correlation and Risk Analysis

We computed Pearson correlations between cyclomatic complexity and other metrics (file size, nesting depth, comment ratio, dependency count) to identify factors associated with higher complexity.

We examine whether functions with high cyclomatic complexity cluster in particular functional categories, revealing where developers invested effort and how complexity was managed.

3.6. Reproducibility

The datasets and analysis notebook are available online in our replication package at this link[6]. The notebook can be executed in Google Colab without local installation. All random seeds are fixed for reproducibility. We cannot share the original or modified source code of *DPaint* in compliance

[6] `https://github.com/giuseppedestefanis/HistoricSoftwareEngineering_replicationPackage`

with the license terms of its release. However, the replication package includes Python scripts that independently verify the design pattern findings reported in Section 4.3 and the quantitative metrics, together with the metric exports and the analysis notebook. We make available upon request a single diff file containing all the changes required to recompile the source code with SAS/C v6.58, which allows reproducing the changes made to the source code to recompile it. We also cannot share a hard-disk file (HDF) with the environment used to recompile the modified source code of *DPaint* again in compliance with the license terms of Amiga Kickstart (ROM), the Amiga Workbench (OS), and the SAS/C compiler. However, we make the HDF available upon request to the first author.

4. Results

We now report the results of applying the previous method and answer our research questions.

4.1. RQ1: How and by whom was DPaint developed, and what was its overall quality?

We obtained the source code from the Computer History Museum's software archive and extracted metrics using Understand SciTools. This produced four datasets: file-level metrics (88 files, 28 metrics each), function-level metrics (546 functions, 21 metrics each), aggregated dependency counts per source file, and file-to-file dependency edges (465 relationships with weights). Table 1 summarises descriptive statistics.

Table 1: Descriptive metrics for the *DPaint* source code

Metric	Min.	Avg.	Max.	Sum
Number of Files	–	–	–	89
Lines of code	10	193	1,337	17,001
Cyclomatic complexity (per file)	0	14	117	1,223
Cyclomatic complexity (per function)	1	2.2	74	–
Nesting (per function)	0	0.5	4	–

- Number of Files: $89 = 77 + 11 + 1$, with 77 C source files, 11 C header files, and `PRISM.txt` that describes the overlays used in *DPaint*.

- Lines of code: these numbers include comments and blank lines. Total is 17,001. Files `INITREAD.C` and `INITWRIT.C` have only 10 LOCs each because they are placeholders used with the overlay system. `PALETTE.C` is the longest file with 1,337 LOC.

- Cyclomatic complexity: it is computed as the sum of the CC of the function in each file. Total is 1,223. Excluding the files with 0 complexity (header files), the min. CC is 1, average 21.8, and max. 117. `PALETTE.C` is the most complex file with 117. These values are computed from file-level metrics, excluding 32 files with zero cyclomatic complexity (header files and files without function definitions), yielding N=56 files.

- Average cyclomatic complexity: `CHPROC.C` has the max. average CC of 8.6 because of its 10 short functions and one very long function, `mainCproc`, which is a long switch statement for keyboard shortcuts.

- Max. cyclomatic complexity: `CHPROC.C` has also the max. CC of 74, again due to its `mainCproc` function.

- Max. Nesting: `CCYCLE.C`, `DPIO.C`, `HSV.C`, `PANE.C`, and `ROT90.C` have the max. nesting of 4, either because they handle user events and possible errors, e.g., `DPIO.C` or `PANE.C`, or perform computations that require decisions at certain limit values.

4.1.1. Contributors and Release Date

Although Dan Silva is credited as the sole author of *DPaint*, source files mention a total of six contributors explicitly, in order of visible importance:

1. Dan Silva, credited as "Dan Silva" in `BITMAPS.C`, "D.Silva" in `PSYM.C`, "DDS" in `ROTATE.C`[7] and "dds" in `DPIFF.C`, "dans" in `MAKEICON.C`[8].

[7]We could not find hard evidence online or elsewhere regarding Silva's possible middle name; however, it is a reasonable assumption given the comment that reads as: `9-8-85 ⇄ DDS ⇄ Ported to Amiga (68000)`.

[8]Similarly to [7], it seems a reasonable assumption given the comment: `REVISION: dans - modified for DPaint - 11/16/85`.

2. Jerry Morrison, credited as "Jerry Morrison" in `UNPACKER.C`, "jhm" in `DPIFF.C`[9].

3. Steve Shaw, credited as "Steve Shaw" in `UNPACKER.C`, "sss" `DPIFF.C`[9].

4. Gordon Knopes, credited in `HSV.C`.

5. An unknown author, credited as "mrp" in `DPIFF.C`.

6. Another unknown author, credited as "tomc" in `MAKEICON.C`.

The earliest known reference dates from 1985/01/30, appearing in the file `system.h`. However, this file originates from Commodore and is attributed to R. J. Mical, as documented below. The earliest date associated with the name of Dan Silva is 1985/06/22, in `BITMAPS.C`. The latest date is 1985/11/07, and it appears in `DPIFF.C`. It is difficult to find the exact day for the commercial release of *DPaint*, often only given as "November 1985", but it must have been after 1985/11/07.

```
/*
 *  system.h, general includer for intuition
 *
 *  Confidential Information: Commodore-Amiga, Inc.
 *  Copyright (c) Commodore-Amiga, Inc.
 *
 *                    Modification History
 * date     author :     Comments
 * ------     ------     ------------------------------------
 * 1-30-85    -=RJ=-     created this file!
 *
*****************************************************************/
```

4.1.2. Development

The file names provided by EA follow the 8+3 naming convention used by Digital Research, Inc. (DRI) CP/M and Microsoft DOS (MS-DOS) in

[9]We could not find hard evidence online or elsewhere regarding Morrison and Shaw's middle names; however, they are reasonable assumptions given that both Morrison and Shaw are credited for writing and contributing to the Interchange Format Files (IFF) standard at EA [33] and must have been the resident experts on all things IFF in EA.

the 1980's and 1990's. Some discussions point to the cross-development of *DPaint* on MS-DOS for the Amiga[10], which supported longer file names of up to 30 characters already at that time. Alternatively, it could have been converted to MS-DOS file names for the production of the final version and the disks. However, we support the former theory because the Amiga disks for *DPaint* use the normal mix of uppercase and lowercase letters expected on the Amiga as well as the necessary commands, library, and scripts to make them bootable. Hence, the disks must have been produced on an Amiga, and there would have been no reason to follow the MS-DOS naming convention but for development.

4.1.3. Compilation

The source code of Deluxe Paint as released by EA in 2015 does not contain any makefile to recompile it "as is". However, it contains the `PRISM.txt` file, shown in Table 2, that provides precious information to recompile it. We reproduce the file as provided with the archive. It shows that:

- *DPaint* was compiled with the *Lattice C* compiler as evidenced by the use of the library `lc.lib`.
- *DPaint* made use of the overlay system provided by the *Lattice C* compiler (and its `blink` linker) and later by the SAS/C compiler (and its `slink` linker).

We further define and discuss overlays when answering RQ4.

4.1.4. Recompilation

We recompiled the source code with the latest version released of the SAS/C compiler for the Amiga, v6.58, the descendant of Lattice C. We could not use a more recent compiler, e.g., *vbcc* v0.9patch3[11], because its linker does not yet support overlays. The goal was not to obtain a modern version of the binary but rather to analyse (1) the completeness of the source files and (2) the quality of the written code.

Starting from the source code, following some external information[12], and using our own knowledge, we could recompile *DPaint* on an Amiga 1200 with

[10] `https://forum.amiga.org/index.php?topic=69530.0`
[11] `http://sun.hasenbraten.de/vbcc/`
[12] `https://forum.amiga.org/index.php?topic=69530.msg793095#msg793095`

Table 2: DPaint overlay description file

```
ROOT astartup.o, prism.o, menu.o, chproc.o, curbrush.o*
paintw.o, modes.o, geom.o, conic.o*
psym.o, lfmult.o, pane.o*
mainmag.o, magops.o, magwin.o, magbits.o*
pgraph.o, bitmaps.o, bmob.o*
blitops.o, maskblit.o, dalloc.o, box.o*
dispnum.o, keyjunk.o, distance.o, mousebut.o, ctrpan.o*
penmenu.o, actbms.o, cursor.o, cursbms.o, dotbms.o*
message.o, ccycle.o, reqcom.o, timer.o, brxform.o
OVERLAY
initovs.o, dpinit.o, dpinit2.o, fontinit.o, bootit.o, hook.o,
fill.o, airbrush.o, random.o, polyh.o, blend.o, shade.o, polyf.o,
    ↪ clip.o, text.o
rot90.o, stretch.o, rotate.o, shear.o, remap.o, bend.o, print.o,
    ↪ spare.o
dopalett.o, palette.o, palbms.o hsv.o
dosymreq.o,symreq.o
dpio.o, fnreq.o, fnbms.o, dpiff.o
*initread.o, ilbmr.o, unpacker.o,iffr.o
*initwrite.o, ilbmw.o, packerf.o, copymem.o, iffw.o, makeicon.o
#
LIBRARY amiga.lib, lc.lib, debug.lib
TO prism
```

Kickstart and Workbench v3.1 with the SAS/C compiler v6.58 and the Cubic IDE v1sp11[13], using both a real computer and the WinUAE emulator[14].

The recompilation required several modifications to the original source code. The primary changes involved updating include paths to match the SAS/C directory structure and adjusting function prototypes to satisfy the stricter type checking of SAS/C v6.58. We also created a Makefile to replace the original batch-style build process described in PRISM.txt. The resulting binary runs on the same hardware and emulator configurations.

The SAS/C compiler reports hundreds of warnings in 11 categories, which

[13] https://www.softwareandcircuits.com/division/amiga/products/cubic/

[14] https://www.winuae.net/

Table 3: SAS/C compiler warnings

Number	Definition
62	constant number out of range for type "*type*" Valid range is *low* to *high*
85	return value mismatch for function "*name*" Expecting "*type1*", found "*type2*"
100	no prototype declared for function "*name*"
132	extra tokens after valid preprocessor directive
154	no prototype declared for function pointer
155	no statement after label
161	no prototype declared at definition for function "*name*"
169	incompatible operands of conditional operator (?:) "*type1*" conflicts with "*type2*"
181	"*name*" was declared both static and external See line *number* file "*filename*"
217	macro invocation may call function multiple times
224	item "*name*" already defined See line *number* file "*filename*"

Table 3 describes. These warnings may not have been produced by the Lattice C compiler (or the cross-compiler used on MS-DOS). They show that the coding discipline/standard was different from "modern" C. In particular, Warnings 85, 100, 154, and 161 concern the lack of (precise) declarations of functions (and function pointers) *before* the definitions/uses of these functions, which was acceptable in pre-ANSI C. Warning 62 warns about the use of `short` to store values greater than 32,767 due to the complicated history of C datatypes and their dependence on platforms (8bit vs. 16bit...) and implementations. Warning 132 is mundane and arises only in EA's IFF header files, as is Warning 155, which happens in a `switch` statement. Warning 169 occurs only once due to the lack of a function pointer declaration (see Warning 154). Warning 181 is due to the lack of function prototypes (see Warning 154). Warning 217 is caused by two macros using the same argument two times in a ternary operator. Warning 224 reports some structures, like `Point`, be defined twice: once in *DPaint*, and again in the SAS/C libraries (possibly due to the difference in compiler versions).

In summary, *DPaint* was released in November 1985 and mainly developed by Dan Silva with contributions from five other developers, primarily on the IFF file format subsystem. The source code comprises 89 files totalling 17,001 lines of code, with a total cyclomatic complexity of 1,223. The code compiles with only warnings (no errors) on a modern SAS/C compiler, indicating high code quality for its era. The 8+3 file naming convention and detailed header comments demonstrate professional development practices despite the absence of modern version control or IDE tools.

Having established the development context and overall quality of *DPaint*, we now examine the architectural styles revealed by its file dependencies.

4.2. RQ2: What architectural styles can be observed in the source code of DPaint?

The subsections below present the results of the graph construction, centrality analysis, community detection, and robustness simulation described in Section 3.2. We first describe the overall structure of the dependency graph and then analyse each subsystem in detail.

To better understand the design and implementation of the source code, we created a visual representation of the core files and their dependencies, shown in Figure 3. This graph, generated from the 465 dependencies identified in our analysis, reveals a design organised into four distinct subsystems, which we grouped manually for visual clarity: Core Files 1, Core Files 2, UI Files, and Utility Files. The greedy-modularity algorithm reported later in this section refines this manual view into five finer communities.

At the heart of the system lie `PRISM.C` and `PRISM.H`, which serve as the central orchestration layer. Our dependency analysis reveals that `PRISM.H` has the highest in-degree (referenced by 48 files), while `PRISM.C` shows significant out-degree (depends on 18 files), confirming its role as core.

The source code contains 11 header files in total, but only four appear as architecturally significant nodes in the dependency network: `PRISM.H` as the central type and macro hub, `GRDEFS.H` for PC-heritage graphics types, `PLBM.H` for bitmap utilities, and `DPIFF.H` for IFF file format types. `DPIFF.C` is classified with core files rather than utility files because it serves as the primary interface between the application and the IFF subsystem, coordinating `ILBMR.C` and `ILBMW.C`.

Table 4 presents the metrics for the main files identified in the dependency graph in the file groups Core Files 1 and 2.

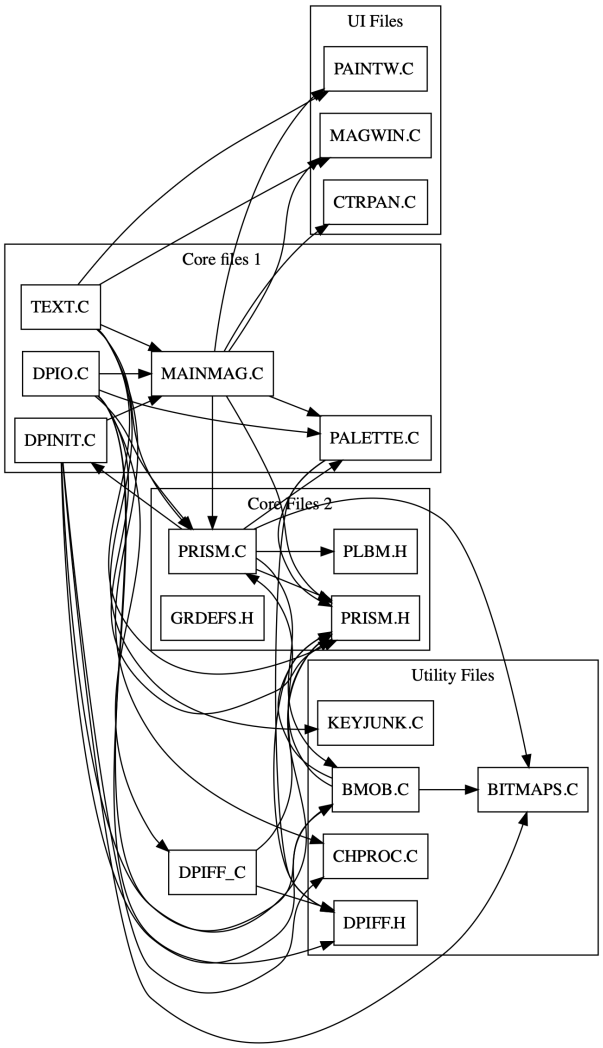

Figure 3: Dependency structure among selected core files of *DPaint*. Arrows denote "depends on" relationships taken from the extracted dependency data, in which `PRISM.C` and `PRISM.H` are the principal hubs. The figure is an illustrative subset of the internal dependency graph (70 files, 366 edges): the boxes group files by role (user interface, core, and utility) as a manual simplification and do not correspond to the five communities identified by the modularity analysis; `SYSTEM.H`, which has the highest in-degree (58) but is a general header aggregating the Amiga OS interfaces, is omitted; and `GRDEFS.H`, a header of macro and type definitions, is shown for context although it carries no extracted dependency edges.

Table 4: Metrics for Core Files of Deluxe Paint

File	*Type*	*LOC*	*Functions*	*Sum CC*	*Dependencies*	*Dependents*	*Comments*
DPINIT.C	Core	439	8	27	19	1	21%
DPIO.C	Core	352	15	41	19	1	9%
GRDEFS.H	Header	331	-	-	0	0	73%
MAINMAG.C	UI	451	32	58	15	14	9%
PALETTE.C	Core	1,337	31	117	2	7	40%
PLBM.H	Header	101	-	-	2	2	89%
PRISM.C	Core	380	26	37	18	27	22%
PRISM.H	Header	415	-	-	1	48	175%
TEXT.C	Core	200	14	27	11	1	8%

Core Files 1 encompasses files handling primary functionality:

- PALETTE.C: Colour palette handling (1,337 LOC, highest complexity).
- DPINIT.C: System initialisation and configuration.
- DPIO.C: Disk I/O operations for loading/saving images.
- MAINMAG.C: Main UI control and magnification features.
- TEXT.C: Text rendering and font management.

These files exhibit tight coupling with Core Files 2, particularly with PRISM.C/H, and depend on the utility module for specialised operations. The UI Files module (CTRPAN.C, MAGWIN.C, PAINTW.C) maintains separation from core logic while interfacing through MAINMAG.C.

The design reveals several principles:

1. **Centralised coordination**: PRISM.C/H acts as the system hub, reducing direct inter-module dependencies. This centralisation is reinforced by the use of *PRISM.H* as a shared header, which provides type definitions and macros to all files without requiring direct inter-file dependencies.

Table 5: Top 10 files by betweenness centrality (left) and in-degree (right) on the internal dependency graph.

File	Betw.	File	In-deg.
PRISM.C	0.137	SYSTEM.H	58
PAINTW.C	0.096	PRISM.H	48
MENU.C	0.074	PRISM.C	27
MODES.C	0.067	PAINTW.C	19
MAINMAG.C	0.043	CURSOR.C	14
DPINIT.C	0.034	CCYCLE.C	14
DPIO.C	0.030	MODES.C	14
PGRAPH.C	0.023	MAINMAG.C	14
BMOB.C	0.020	BMOB.C	13
CURBRUSH.C	0.017	PGRAPH.C	13

2. **Layered abstraction**: Utility files provide low-level operations, core files implement business logic, and UI files handle presentation.

3. **Modular design**: Despite the constraints of C and 1980s development practices, the system exhibits clear module boundaries.

4.2.1. Centrality Analysis

Table 5 reports the top files by betweenness centrality and in-degree on the internal dependency graph (70 nodes, 366 edges, excluding external Amiga OS headers).

PRISM.C has the highest betweenness centrality (0.137), confirming its role as the central coordinator through which most inter-file communication flows. SYSTEM.H has the highest in-degree (58), reflecting its role as the universal system header included by most files. Among non-header files, PRISM.C (in-degree 27) and PAINTW.C (in-degree 19) are the most depended-upon implementation files.

4.2.2. Community Detection

Greedy modularity optimisation on the undirected projection of the internal dependency graph identifies five communities (modularity $Q = 0.191$):

1. **System core** (17 files): PRISM.C, DPINIT.C, MENU.C, PANE.C, BITMAPS.C, CTRPAN.C, and initialisation/utility files. This community centres on the application lifecycle and UI framework.

2. **Drawing utilities** (16 files): `SYSTEM.H`, `GEOM.C`, `FILL.C`, `CONIC.C`, `CLIP.C`, and geometric/requester files. This community handles drawing primitives and user requests.

3. **Interaction and rendering** (15 files): `PAINTW.C`, `MODES.C`, `PGRAPH.C`, `CHPROC.C`, and magnification/symmetry files. This community handles the interactive painting loop.

4. **Bitmap operations** (13 files): `PRISM.H`, `BMOB.C`, `BLITOPS.C`, `CURBRUSH.C`, and transformation files. This community centres on bitmap manipulation and brush transforms.

5. **File I/O and colour** (9 files): `PALETTE.C`, `MAINMAG.C`, `DPIO.C`, `DPIFF.C`, and IFF-related files. This community handles file operations and colour management.

The five communities broadly correspond to functional subsystems, refining the four manual subsystems of Figure 3 into finer groups, though the relatively low modularity (0.191) reflects the tight coupling inherent in a 256 KB application where most files share a common header (`PRISM.H`).

4.2.3. Robustness Analysis

Progressive removal of the ten highest-ranked files reveals contrasting fragmentation patterns under the two targeting strategies. Removal by betweenness centrality (targeting architectural bridges) produces gradual degradation: after removing 10 files, the graph fragments into 5 components, with the largest retaining 78.6% of nodes. Removal by in-degree (targeting foundational files) causes rapid fragmentation: after 10 removals, the graph splits into 18 components, with the largest retaining only 60.0% of nodes.

This asymmetry indicates that *DPaint* architecture is resilient to the loss of bridge files (whose routing function can be partially absorbed by alternative paths) but sensitive to the loss of foundational files such as `SYSTEM.H` and `PRISM.H`, whose removal immediately isolates dependent files. This pattern is consistent with a hub-dependent architecture where a small number of highly connected files provide essential services to the entire source code.

We selected four files for detailed analysis based on their centrality and architectural significance: `PRISM.C`, `PRISM.H`, `GRDEFS.H`, and `PLBM.H`. These files collectively define the fundamental data structures, constants, and core routines that enable Deluxe Paint functionalities.

PRISM.H. This header file serves as the central definition hub, with our analysis showing that it is referenced by 48 other files. It defines critical data types, constants, and macros used throughout the source code. The high comment ratio (175%) indicates the developers' recognition of its importance for system understanding.

One of the central data structures defined here is the BMOB (masked bitmap object):

```
typedef struct {
    BoxBM pict,save;
    UBYTE *mask;
    SHORT xoffs,yoffs;
    UBYTE flags;
    UBYTE xpcolor;
    UBYTE minTerm;
    UBYTE planeUse,planeDef;
    } BMOB;
```

The BMOB structure encapsulates a bitmap with its mask, offsets, and rendering attributes. This compact 16-byte structure (on 16-bit systems) enables efficient bitmap manipulation, central to drawing operations. Its implementation is as follows:

- `pict` and `save`: two independent `BoxBM` structures within each `BMOB`, containing bitmap data and dimensions. The dual structure supports undo operations: `pict` holds the current bitmap data, while `save` preserves a snapshot of the background for restoration.
- `mask`: Pointer to transparency mask data. Each byte determines pixel opacity (0 = transparent, non-zero = opaque), enabling efficient selective drawing without per-pixel alpha calculations.
- `xoffs`, `yoffs`: 16-bit position offsets for efficient placement without modifying pixel data.
- `flags`: Bit field storing boolean states in a single byte, demonstrating memory optimisation.
- `xpcolor`: Transparent colour index for colour-key transparency, an alternative to mask-based transparency.

- `minTerm`: Blitter operation code. The Amiga's Blitter performed 256 different logical operations between source and destination; storing this per-bitmap optimised rendering paths.
- `planeUse`, `planeDef`: Bitplane configuration for the Amiga's planar graphics architecture.

The file also defines the drawing modes and tool interactions through enumerated constants and mode tables:

```
#define Mask          0
#define Color         1
#define Replace       2
#define Smear         3
#define Shade         4
#define Blend         5
#define CyclePaint    6
#define Xor           7

#define IM_none       0xff
#define IM_null       0
#define IM_shade      1
#define IM_draw       2
#define IM_vect       3
...
```

Hardware abstraction appears in Blitter operation macros:

```
#define BlitSize(bpr,h) ((h<<6)|(bpr>>1))
#define REPOP 0xCC
#define NOTOP 0x33
#define XOROP 0x66
...
```

The `BlitSize` macro encodes both height and bytes-per-row into a single 16-bit value using bit shifting, matching the hardware register format of the Blitter. This direct hardware mapping eliminates conversion overhead in performance-critical paths.

PRISM.C. This file implements the core logic, including the main entry point and primary event loop. Despite containing only 26 functions for a total of

380 lines, it orchestrates the entire system:

```
main(argc,argv) int argc; char *argv[]; {
    DPInit(argc,argv); /* initialization code */
    NewIMode(IM_draw);
    SetAirBRad(PMapX(24)); /* also brings in the drawing overlay*/
#ifdef DOWB
    if (loadAFile) LoadPic();
#endif
    CopyWNotice();
    loaded = YES;
    PListen();       /* this is the program */
    if (!haveWBench) BootIT();
    else CloseDisplay();
    }
```

The modular design delegates initialisation to `DPInit()`, mode management to `NewIMode()`, and the main event loop to `PListen()`. This separation of concerns enhances maintainability despite the procedural paradigm.

Memory management appears in `AllocTmpRas()`:

```
AllocTmpRas() {
    int  plsize = mainRP->BitMap->BytesPerRow*mainRP->BitMap->Rows;
    DFree(tmpRas.RasPtr);
    tmpRas.RasPtr = (BYTE *)ChipAlloc(plsize);
    if (tmpRas.RasPtr==NULL) InfoMessage(" Couldnt alloc", "tmpRas.
        ↪ ");
    tmpRas.Size = plsize;
    mainRP->TmpRas = &tmpRas; /* temp raster for area filling*/
    }
```

The use of `ChipAlloc()` rather than standard `malloc()` demonstrates hardware-aware programming. The Amiga dual memory architecture required graphics data in Chip RAM (accessible to custom chips) vs. Fast RAM (CPU-only). This 256 KB chip RAM constraint demanded careful allocation strategies. *DPaint* uses Fast RAM implicitly for all non-graphics data: code, stack, and heap allocations via the standard `AllocMem()` without the `MEMF_CHIP` flag. Only graphics buffers (bitmaps, sprites, copper lists) require Chip RAM. The `DAlloc()` function in `DALLOC.C` allocates from general memory (Fast RAM when available), while `ChipAlloc()` explicitly requests

Chip RAM via `MEMF_CHIP`.

The temporary raster serves flood-fill operations, which require tracking visited pixels. By maintaining a dedicated buffer, the algorithm avoids modifying the source image during fills, preventing artefacts and enabling cancellation. The single allocation strategy also minimises fragmentation in the limited chip RAM space.

GRDEFS.H. This graphics definitions header (73% comment ratio) demonstrates space-efficient memory optimisation through bit fields:

```
typedef struct {
    unsigned border : 1; /* draw border */
    unsigned always : 1; /* call even if mouse didnt move */
    unsigned hangOn : 1; /* keep control while button down */
    } WFlags;
```

Each flag occupies exactly one bit, packing three boolean values into 3 bits versus 12 bytes with standard integers. On memory-constrained systems where this structure might be instantiated hundreds of times for UI elements, the savings are substantial. The `MenuItem` structure (defined in `GRDEFS.H`) is instantiated for every menu item, toolbar button, and dialogue control. The control panel alone (`CTRPAN.C`) defines approximately 40 UI elements. With the bit-field packing, each element saves 9 bytes compared to using standard integers, yielding roughly 360 bytes saved, meaningful on a system with 256 KB total RAM.

The file defines pixel manipulation macros using bit operations:

```
#define xword(x)  (((x)>>LGPPB)&0xfffe)
#define xbyte(x)  (x>>LGPPB)
#define wrdsrqd(w) ((w+PPWM1)>>LGPPW)
#define colForY(y) colw = curpat[y&3]
```

These macros compute memory addresses for pixel access. The shift operations (`»`) divide by powers of two more efficiently than division operators. The `colForY` macro implements pattern lookup using bitwise `&` for fast modulo operation, essential for real-time pattern fills.

PLBM.H. This header defines the Deluxe Paint file format based on EA's IFF standard. The chunk-based architecture enabled extensibility while maintaining compatibility:

```
typedef struct {
    ID      ckID;
    LONG    ckSize;
    } ChunkHeader;

typedef struct {
    UWORD rowWords;       /* # words in an uncompressed row */
    UWORD w, h;           /* raster width & height in pixels */
    UBYTE depth;          /* # destination bitplanes */
    UBYTE nPlanes;        /* # source bitplanes (BITP chunks) */
    Masking masking;      /* masking technique */
    PixelFormat pixelFormat;/* pixel size and aspect */
    Compression compression;
    UBYTE pad1;           /* UNUSED. For consistency, put 0 here.*/
    UWORD transparentColor; /* transparent "color number" */
    } BitMapHeader;
```

The format's efficiency stemmed from matching the Amiga planar graphics. Rather than storing pixels sequentially (chunky format), PLBM interleaves bitplanes, allowing direct Blitter operations without conversion. This design traded some flexibility for significant performance gains.

Type-safe enumerations enhance code clarity:

```
typedef enum {mskNone = 0, mskHasMask = 1, mskHasTransparentColor =
    ↪ 2}
    _Masking;
typedef UBYTE Masking;
```

This pattern provides symbolic names while controlling storage size, balancing expressiveness with memory efficiency.

Analysis Summary. These four files reveal recurring engineering practices:

1. **Hardware awareness**: Direct mapping to Blitter operations, Chip RAM allocation, and planar graphics.
2. **Memory optimisation**: Bit fields, compact structures, and careful allocation strategies.
3. **Performance focus**: Bitwise operations, macro preprocessing, and minimal abstraction overhead.

4. **Maintainability**: High comment ratios in critical files, clear naming conventions, and modular organisation.

The source code shows that even within severe constraints, disciplined design can produce systems that are efficient, capable, and maintainable.

4.2.4. Other Files of Interest

Based on the source code dependency graph and the metrics analysis, several other files merit analysis in more detail. These files play important roles in the *DPaint* source code, either because of their complexity, size, or the number of dependencies they have.

- `PALETTE.C`: This file has a high cyclomatic complexity (117) and a large number of lines of code (1337), which suggests that it contains complex logic related to colour palette management. It also has a relatively high ratio of comments to code (0.4), which indicates that the code is well-documented and provides detailed information about the colour system used by Deluxe Paint.

- `DPINIT.C`: This file seems to be responsible for initialising various parts of the Deluxe Paint system, based on its name and its dependencies on other modules, like `BITMAPS.C`, `BMOB.C`, and `PRISM.H`. It has a fairly high cyclomatic complexity (27) and a large number of lines of code (439), which suggests that the initialisation process is complex and involves many different subsystems.

- `DPIO.C`: This file handles disk I/O operations for Deluxe Paint, based on its name and its dependencies on modules like `DPIFF.H` (which defines the ILBM file format) and `FNREQ.H`.

- `MAINMAG.C`: This file seems to be a main control module for the Deluxe Paint UI, based on its name and its dependencies on modules, like `CTRPAN.C` (control panel), `MAGWIN.C` (magnification window), and `PAINTW.C` (paint window). It has a fairly high cyclomatic complexity (58) and a large number of code lines (451), suggesting that it contains complex logic to coordinate the various elements and tools of the UI.

- `TEXT.C`: This file handles text rendering and editing operations. It has dependencies on several key modules, like `CHPROC.C` (character processing), `KEYJUNK.C` (key event handling), and `PRISM.H`, which suggests that it is closely integrated with the core Deluxe Paint system.

PALETTE.C. The largest file in the source code (1,337 LOC, Sum CC=117) implements the complete colour palette editor, including RGB and HSV sliders, colour cycling, spreading, and copy/exchange operations. It manages its own floating Intuition window with 31 functions. The high sum complexity reflects the breadth of user-facing operations rather than algorithmic intricacy, with an average per-function complexity of 3.8.

DPINIT.C. The initialisation module (439 LOC, 8 functions, Sum CC=27) sets up the screen, windows, views, and application state. It configures display modes, allocates Chip RAM resources, and processes command-line parameters. The file demonstrates the complexity of initialising a graphics application on the Amiga, including detection of available memory (256 KB vs. 512 KB) and negotiation with the operating system for shared resources.

DPIO.C. The file I/O module (352 LOC, 15 functions, Sum CC=41) handles loading and saving images in IFF/ILBM format. It coordinates with the overlay system to manage the single-floppy-drive constraint, strategically freeing `TmpRas` before displaying file requesters (an instance of the cooperative resource sharing idiom described in Section 4.4). The file also manages brush loading/saving and format conversion between internal and external representations.

MAINMAG.C. The magnification and undo module (451 LOC, 32 functions, Sum CC=58) implements the two-buffer undo system and the magnification window. `UndoSave()` copies changed regions from the screen bitmap to `hidbm`, while `SwapHidScr()` performs the triple-XOR undo swap requiring zero additional memory. The magnification subsystem tracks dirty regions and propagates updates between the main canvas and the zoomed view.

TEXT.C. The text rendering module (200 LOC, 14 functions, Sum CC=27) provides interactive text input directly on the canvas. It manages cursor positioning, blinking, font selection via the Amiga `diskfont.library`, and keyboard input processing. Text is rendered character-by-character using the graphics context stack (`PushGrDest`/`PopGrDest`) described in Section 4.4.

The dependency analysis reveals a hub-and-spoke architecture centred on `PRISM.C` and `PRISM.H`, with five communities identified through greedy modularity optimisation: a system core, drawing utilities, interaction and rendering, bitmap operations, and file I/O with colour management. The design achieves modularity through centralised coordination rather than strict layering: `PRISM.H` acts as a shared vocabulary, while `PRISM.C` orchestrates control flow across subsystems. The architecture is resilient to targeted removal of bridge files but sensitive to the loss of hub files, consistent with a scale-free network topology. This modular organisation, achieved without formal architectural guidelines, suggests that experienced developers naturally converge on well-structured designs when building complex interactive systems.

The hub-and-spoke architecture identified above raises the question if the code exhibits recognisable design patterns at a finer structural level.

4.3. RQ3: What design patterns are present in the source code of DPaint?

In this article, *patterns* refers to structural, behavioural, and creational solutions that have GoF analogues, while *idioms* (Section 4.4) refer to platform-specific architectural solutions with no GoF counterpart. Both emerged organically from the engineering constraints of the Amiga platform.

To complement the graph-theoretic analysis of the dependency network, we conducted a systematic manual inspection of the source code to identify structural, behavioural, and creational constructs that correspond to design patterns later formalised by Gamma et al. [30].

Every pattern identification reported below is grounded in specific file names, function signatures, struct definitions, or code idioms observed directly in the source. We distinguish between patterns that arise from deliberate abstraction and those that emerge from hardware necessity, and we note where a pattern is partial or degenerate relative to its GoF descendant.

The source code predates the Gang of Four's catalogue by nine years and was written in pre-ANSI C (the dialect predating the C89/ANSI standard) without access to object-oriented language features. Despite this, we identify antecedents to *eleven* GoF patterns spanning three families, Structural (3), Behavioural (5), and Creational (3), plus seven additional architectural patterns that have no direct GoF analogue but reflect the constraints of the target platform.

4.3.1. Structural Patterns

Façade. Four distinct Façades mediate access to lower-level subsystems.

The most architecturally significant is `PGRAPH.C`, which provides a unified drawing API (`PWritePix`, `PHorizLine`, `PFillBox`, `PFloodArea`) over the Amiga's `RastPort` (the graphics drawing-context structure) and Blitter hardware. A four-deep push/pop context stack (`PushGrDest`/`PopGrDest`, lines 38–52) manages the current rendering target, clipping rectangle, and canvas bitmap, allowing the magnification window, control panel, and main canvas to share drawing code while targeting different destinations. `SetPaintMode()` (line 104) atomically translates the eight application-level paint modes into Amiga-specific `DrawMode` values, pixel-writer function pointers, and Blitter Minterms (basic transformations), which the Amiga API does not provide.

`BLITOPS.C` and `MASKBLIT.C` wrap the Amiga Blitter's memory-mapped registers (cast from address `0xDFF040` to a typed C struct, `BlitterRegs`) behind callable functions. `MaskBlit()` (lines 52–154) encapsulates a hardware bug (the "Destination Wrap-Around Bug") by recursively splitting oversized blits into two halves, a workaround entirely invisible to callers. `BITMAPS.C` wraps per-plane Chip RAM allocation[15] with a global memory-floor guard (`MINAVAILMEM` = 13,000 bytes) and lazy reallocation logic. `PANE.C` implements a lightweight sub-window manager within a single Amiga Intuition window, providing a linked list of `Pane` structs with polymorphic callbacks (`charProc`, `mouseProc`, `paintProc`), effectively replacing multiple Intuition windows that would be prohibitively expensive in memory.

In the GoF, Façade provides a unified interface to a set of interfaces in a subsystem; *DPaint*'s Façades go further by adding state management (the graphics context stack), automatic clipping, and hardware-bug encapsulation—responsibilities absent from the underlying Amiga OS API.

Adapter. Two adapters are present in *DPaint*, with a third evidenced at design time.

First, the virtual/physical coordinate system defined by macros `PMapX`, `PMapY`, `VMapX`, and `VMapY` (`PRISM.H`, lines 110–115) translates between a resolution-independent 640×400 virtual coordinate space and the hardware-specific physical pixel grid. The mapping is controlled by shift values `xShft` and `yShft`, which are set per display mode, using bit-shift operations—the

[15] (The Amiga stores bitmaps as separate bitplanes, each a one-bit-per-pixel buffer.)

fastest possible transformation on the MC68000's slow multiplier. This is a textbook Adapter: two incompatible coordinate interfaces (logical vs. physical) are bridged by a translation layer.

Second, the IFF file format bridge (`DPIFF.C`, `ILBMR.C`, `ILBMW.C`) uses the `MaskBM` struct (`DPIFF.H`, lines 26–38) to mediate between the external IFF/ILBM representation (24-bit RGB triples, `BitMapHeader`, `GroupContext`) and the internal Amiga-native representation (12-bit packed colour words, `struct BitMap`, `BMOB`). Colour conversion code in `ILBMR.C` (lines 29–31) and `ILBMW.C` (lines 66–68) explicitly adapts between the two colour encodings.

A third, design-time adapter is evidenced by the parallel type systems in `GRDEFS.H` (PC PRISM heritage: `Bitmap`, `DOB`, `BMObj`, `Window`) and `PRISM.H` (Amiga: `BoxBM`, `BMOB`, `Pane`). Both define structurally isomorphic types (`Point`, `Box`, `Dims`) with identical field names but different underlying sizes (`int` vs. `SHORT`) and different bitmap representations (PC segment:offset vs. Amiga planar `BitMap`). No runtime bridging code exists; the adaptation was performed at porting time.

The Adapter pattern in *DPaint* shows that interface translation is a fundamental engineering need that predates its formal cataloguing. The coordinate mapping macros and IFF bridge serve exactly the purpose that the GoF describes (decoupling clients from incompatible interfaces) through compile-time macros and procedural conversion rather than wrapper classes.

Decorator. The GoF Decorator pattern attaches additional responsibilities to an object at runtime by wrapping it. Three instances in *DPaint* exist, using C struct embedding.

The `BMOB` type system exhibits a three-layer progressive wrapping: `Box` (pure geometry) → `BoxBM` (adds a `BitMap` pointer—spatial positioning of a bitmap) → `BMOB` (adds a save-under buffer, transparency mask, grab-point offsets, display flags, Blitter Minterm, and per-plane selection controls). Each layer adds orthogonal capabilities without modifying the underlying layer. The function `CopyBoxBM()` operates at the `BoxBM` level; `PaintOb()` (`BMOB.C`, lines 204–234) utilises all `BMOB`-level fields for mode-dependent rendering through the `paintProps[]` dispatch table.

The menu item type system in `GRDEFS.H` (lines 236–289) uses the same first-field embedding idiom: seven variant types (`Labl`, `ColBx`, `Butn`, `TBox`, `Mar`, `ScBar`, `NBox`) all embed `MenuItem hdr` as their first field with a `type` discriminant enabling runtime dispatch. This is effectively manual polymorphism via tagged structs.

Finally, the `IModeDesc.flags` bitfield (`PRISM.H`, lines 306–333) uses fourteen orthogonal bit flags (`PERM`, `NOGR`, `NOBR`, `NOSYM`, `SYMUP`, `SYMDN`, `SLAVE`, `NOLOCK`, `EVTIM`, `PNTWDN`, `COMPLETE`, `RIGHTBUT`, `NOERASE`, `NOCON`) to compose behaviours. The master interaction-mode table (`PAINTW.C`, lines 237–271) generates 32 distinct modes from flag combinations, achieving the Decorator's goal—composing behaviours orthogonally—using 14 bits rather than a class hierarchy.

DPaint's use of struct embedding and bitfield composition achieves the Decorator's goal of attaching additional responsibilities without modifying existing structures. The three-layer BMOB hierarchy and the 14-bit IMode flag field are functionally equivalent to Decorator chains, showing that the pattern's intent (orthogonal composition of capabilities) can be realised without class inheritance.

4.3.2. Behavioural Patterns

Command. Operations in *DPaint* are encoded as storable, swappable function pointers rather than being executed in-line. The core mechanism is the *eight-slot architecture*: eight `local void (*proc)()` variables (`upShow`, `upMove`, `upClear`, `WentDn`, `dnShow`, `dnMove`, `dnClear`, `WentUp`) in `PAINTW.C` (lines 28–30) encode the complete interactive behaviour of a drawing mode.

The function `IModeProcs()` (line 357) atomically fills in the eight slots. `NewIMode()` (line 292) resets all to `nop` before invoking the new mode's `startProc`, which points to the appropriate function. The operations are stored (parameterised) and later invoked by the event loop upon mouse events or menu item selection—precisely the GoF Command intent of encapsulating a request as an object.

Menu actions are encoded in dispatch tables: `picProc[]`, `prefsProc[]`, and `MenProcs[]` (`MENU.C`, lines 159, 364, 391) are arrays of function pointers indexed by menu item number, providing $O(1)$ dispatch.

The undo system uses a degenerate Command variant: the `didType` variable declared in `MAINMAG.C`, line 71, and the constants in `PRISM.H`, lines 396–399, record what kind of operation was performed (`DIDNothing`, `DIDClear`, `DIDMerge`, `DIDHPoly`), and `Undo()` dispatches on this variable to select the correct undo.

The latter undo is a single-level Command with a serialised type tag instead of a full command-object stack—a pragmatic compromise in *DPaint* given the memory constraint that precludes maintaining a history. Beyond the `didType` mechanism, the undo system interacts with the two-buffer archi-

tecture (described in Section 4.4): `UndoSave()` commits the current drawing state by copying the changed region from the screen bitmap to `hidbm`, while `Undo()` swaps the two via the triple-XOR Blitter trick. This tight coupling between the Command tag and the Prototype-based buffer swap illustrates how multiple patterns can collaborate within severe memory constraints.

Strategy. Algorithms are selected at runtime via function pointers in three distinct occurrences.

First, the pixel-writing strategy table `wrPixTable[]` (`PGRAPH.C`, lines 100–102) maps the eight paint modes to per-pixel functions: `PWritePix` (normal), `PSmearPix` (smear), `PShadePix` (shade), `PBlendPix` (blend). `SetPaintMode()` loads the active strategy into the global `CurPWritePix`, which all geometric primitives invoke. A parallel table `paintProps[]` (`BMOB.C`, lines 172–181) selects Blitter configurations per mode, creating a *two-dimensional strategy space*: pixel-combination mode × Blitter configuration.

Second, global function pointers `doit` and `xorit` (`MODES.C`, line 96) hold the current geometric drawing operation. Each mode's `startProc` assigns the concrete operation (e.g., `doit = &vec` for vectors, `doit = &cir` for circles), enabling the rendering functions (`PaintIt`, `PaintSym`, and `FlipIt`) to be invoked without knowing which shape is active.

Third, geometric primitives accept per-pixel strategy as parameters via a function pointer `fun`: `PLineWith(x1, y1, x2, y2, fun)` (`GEOM.C`, line 20), `PCircWith(xc, yc, rad, func)` (line 103), and `PEllpsWith(xc, yc, a, b, func)` (`CONIC.C`, line 193). The same Bresenham or conic-section algorithm is reused with different pixel-level strategies, avoiding the code duplication that would result from hardcoding modes into each primitive.

In the GoF, a Strategy defines a family of algorithms and makes them interchangeable, which *DPaint* implements without classes by using function pointers at three levels of granularity: system-wide (paint mode), per-interaction (geometric shape), and per-primitive (pixel function parameter).

State. The interaction mode (`IMode`) constitutes an explicit state machine.

The `imodes[NIMODES]` table (`PAINTW.C`, lines 237–271) is a 32-entry dispatch table in which each `IModeDesc` entry specifies behavioural flags, cursor type, activity indicator, a `nextIMode` for chaining, and a `startProc` that reconfigures the system.

State transitions are managed by four functions: `NewIMode()` enters a state by resetting all command slots and invoking `startProc`; `NextIMode()`

chains to the successor state or reverts; `RevertIMode()` returns to the last permanent mode; `AbortIMode()` cancels and reverts. The `nextIMode` field creates multi-step interaction chains (e.g., `IM_curve1` → `IM_curve2` → revert; `IM_poly` → `IM_poly2` → `IM_poly2` looping until abort).

Each mode transition reconfigures the cursor, symmetry settings, grid behaviour, and all eight command slots, making the system "appear to change its class" in GoF terminology. A nested sub-state machine governs constraint behaviour (`Constrain()`, `PAINTW.C`, lines 367–402): four explicit states (`CONS_NOT`, `CONS_INIT`, `CONS_HORIZ`, `CONS_VERT`) with transitions based on the Shift key and initial movement direction.

The IMode state machine is perhaps the closest to a textbook GoF implementation found in *DPaint*. The 32-entry dispatch table with function pointers, chained transitions, and complete behavioural reconfiguration on each state change mirrors the GoF State pattern almost exactly; the only divergence is the use of a table-driven approach rather than state objects.

Observer. Event notification occurs through several mechanisms.

The pane (`PANE.C`) registers `charProc`, `mouseProc`, and `paintProc` callbacks at creation time; the main event loop `PListen()` (line 261) reads Intuition messages and dispatches them to the active pane's handlers. `PaneRefresh()` (line 101) broadcasts repaint notifications to all overlapping Panes—a one-to-many notification.

The magnification context's `updtProc` callback (`PRISM.H`, line 239) is invoked by `UpdtSibs()` whenever drawing modifies a region, propagating change notifications from the drawing subsystem to the magnification renderer. Additional callback hooks include `periodicCall` (`PAINTW.C`, line 364) for periodic operations, such as text-cursor blink, and `tempChProc` (`CHPROC.C`, line 42) for pluggable keyboard handling.

At the hardware level, colour cycling (`CCYCLE.C`, lines 99–120) installs a vertical-blank interrupt handler via `AddIntServer(5, &intServ)`, running at 50Hz (PAL/SECAM) or 60Hz (NTSC) asynchronously from the main program. A `NoCycle` lock flag (line 73) prevents race conditions between the interrupt handler and main-thread palette modifications.

The GoF Observer defines a one-to-many dependency where subjects notify observers of state changes. *DPaint* achieves this via function-pointer registration (pane, `updtProc`, `periodicCall`) and hardware interrupt subscription (VBlank), without a formal subject/observer interface.

Template Method. The GoF Template Method defines the skeleton of an algorithm in a base class, deferring specific steps to subclasses. *DPaint* implements this pattern using fixed procedural skeletons with function pointers for specific steps.

The main mouse handler `mainMproc()` (`PAINTW.C`, lines 407–427) defines a fixed sequence: coordinate transform → grid snap → constrain → display coordinates → dispatch event → periodic callback. The subroutine `GoDown()` (line 166) defines a fixed lifecycle: clear feedback → set state → save undo → *call function pointer* → show feedback; `GoUp()` (line 195) follows a complementary skeleton: clear feedback → set state → *call function pointer* → finalise → advance mode.

The pre-fabricated mode templates in `MODES.C` (`PaintWith`, `VTypeOps`, `XHTypeOps`, `CircTypeOps`, `FCircTypeOps`, lines 121–158) define fixed eight-slot wiring patterns, accepting only the specific geometric function as a parameter. Each concrete mode is then a single line: `void IMVect() { VTypeOps(vec, eraseFB, vec); }`.

The symmetry iterator `SymDo()` (`PSYM.C`, lines 99–131) provides another instance: a fixed skeleton (save points → for each rotation, transform coordinates → call drawing procedure → optionally mirror → restore points) with the drawing operation supplied as a function pointer.

The Template Method instances in *DPaint* illustrate how higher-order functions in C can achieve the same decoupling as virtual methods in an object-oriented language. The `GoDown`/`GoUp` skeletons and the mode template functions fix the algorithm's structure while allowing specific steps to be supplied at runtime through function pointers.

4.3.3. Creational Patterns

Factory Method. Several functions serve exclusively as factories.

`DAlloc()` (`DALLOC.C`, line 18) wraps `AllocMem` by prepending a four-byte size header, enabling `DFree()` without requiring callers to track allocation sizes—a necessity because AmigaOS `FreeMem` demands the exact allocation size. `ChipAlloc()` (line 29) further constrains allocations to Chip RAM via the `MEMF_CHIP` flag.

`BITMAPS.C` provides a hierarchy of bitmap factories: `TmpAllocBitMap()` (line 40) allocates per-plane Chip RAM with rollback on failure; `AllocBitMap()` (line 60) adds a memory-floor check; `NewSizeBitMap()` (line 82) adds lazy reallocation (skipping if dimensions match); `MakeEquivBM()` (line 113) clones geometry from an existing bitmap. Callers specify only depth, width, and

height; word alignment, plane-size calculation, and Chip RAM constraints are encapsulated.

`ExtractBMOB()` (`BMOB.C`, lines 239–268) orchestrates multi-step BMOB construction: bitmap allocation, bounding-box setup, mask allocation, pixel copy via Blitter, mask generation from transparent colour, and control-field initialisation. `SelPen()` (`CURBRUSH.C`, line 141) is a parameterised factory dispatcher that decodes a packed integer encoding (pen type in upper nibble, size in lower 12 bits) and routes to specialised constructors (`RoundPen`, `SquarePen`, `DotsPen`, `OneBitPen`), each sharing a common finalisation protocol (`FixUpPen()`).

In the GoF formulation, Factory Method defers instantiation to subclasses. *DPaint* achieves the same decoupling via dispatch tables and parameterised functions rather than class hierarchies.

Singleton/Monostate. The Singleton pattern, or more precisely the Monostate, is exemplified by `PRISM.C`, which defines approximately 60 global variables constituting the entire application state: display resources (`mainW`, `screen`, `mainRP`, `vport`), the hidden bitmap (`hidbm`), rendering temporaries (`tempRP`, `screenRP`, `tmpRas`), the current brush and pen objects (`curbr`, `curpenob`), the spare canvas (`sparebm`), and scratch bitmaps (`tmpbm`, `tmpbm2`). Strictly speaking, this is a Monostate rather than a Singleton: there is no access-controlled `getInstance()` method, but the effect is identical, namely a single set of shared state accessed by all files. In C, where there are no classes, the distinction between Singleton (controlled instantiation) and Monostate (shared state) collapses into the same implementation, file-scope globals. Every other file accesses these through `extern` declarations. This is the Singleton pattern in its purest C form: the module is the singleton, and its file-scope globals are the shared state.

The single-instance design is not merely convenient—it is enforced by hardware. Chip RAM is physically sufficient for only one screen bitmap, one hidden bitmap, one brush, one pen, and one temporary raster. The `PointRec pnts` structure (`PAINTW.C`, line 33) with macro aliases `mx`, `my`, `sx`, `sy` (`PNTS.H`) centralises coordinate state for the single mouse input device. The pointer `curob` (`MODES.C`, line 46) always references either `curpenob` or `curbr`—a singleton active-tool pointer switched by `UsePen()`/`UseBrush()`.

The Singleton/Monostate pattern in *DPaint* is unique among the identified patterns because it is the only one that is not merely motivated but physically enforced by the hardware. The GoF Singleton controls instantia-

tion through a class method; *DPaint*'s Singleton is enforced by the 256 KB Chip RAM ceiling, which physically cannot accommodate duplicate screen buffers, bitmaps, or raster structures. This suggests that the Singleton pattern may have its deepest roots not in software design philosophy but in the material constraints of the hardware on which software runs.

Prototype. Object creation by cloning existing instances exists in *DPaint*.

`DupBitMap()` (`BITMAPS.C`, line 127) performs a shallow clone (shared bitplane data); `CopyBitMap()` (line 133) performs a deep clone via Blitter `BltBitMap`. `CopyBMOBPict()` (`BRXFORM.C`, lines 29–41) deep-clones a BMOB's picture data (bitmap, mask, bounding box, offsets, transparent colour). `SaveBrush()` (line 44) clones the brush into `origBr` before transformations; `RestoreBrush()` (line 57) moves the data back. This applies the Prototype pattern as a checkpoint for non-destructive operations: the brush is saved into `origBr` before a transformation such as rotate, stretch, or shear, and restored if the user cancels, creating a restorable snapshot analogous to a database checkpoint.

The undo system is fundamentally Prototype-based: `hidbm` holds a persistent clone of the canvas state, and `SwapHidScr()` (`MAINMAG.C`, lines 102–110) performs an in-place swap via three Blitter XOR passes (`CopyFwd(XOROP); CopyBack(XOROP); CopyFwd(XOROP)`)—an operation requiring no additional memory. Brush selection from the canvas (`DoSelBr()` → `ExtractBMOB()`) copies a rectangular region of the canvas bitmap into the brush via Blitter, creating the brush by cloning existing pixels.

4.3.4. GoF Pattern Correspondence

Table 6 provides a detailed comparison between the patterns observed in *DPaint* and their GoF formalisations. For each GoF pattern, we identify the specific *DPaint* mechanism, the key structural divergence from the GoF specification (typically the absence of class hierarchies and virtual dispatch), and the hardware constraint that plausibly motivated the design.

Table 6: Correspondence between observed *DPaint* idioms and GoF design patterns. For each pattern, we list the primary evidence location, the *DPaint* implementation mechanism, and the key divergence from the GoF specification.

GoF Pattern	Evidence	DPaint Mechanism	Divergence from GoF
Structural			
Façade	`PGRAPH.C`	Unified drawing API with context stack, clipping, and mode translation over Amiga `RastPort` and Blitter	Adds state management and bug encapsulation beyond simple interface unification
Façade	`BLITOPS.C`, `MASKBLIT.C`	Typed struct overlay on Blitter registers; hardware-bug workaround hidden from callers	Wraps hardware registers, not a software subsystem; includes cooperative arbitration
Adapter	`PRISM.H:110-115`	Bit-shift macros translating virtual / physical coordinates per display mode	Compile-time macro adapter, not a wrapper class; shift values set at initialisation
Adapter	`DPIFF.C`, `ILBMR/W.C`	`MaskBM` struct bridging IFF format types and internal `BitMap`/`BMOB`	Bidirectional procedural conversion, not a class wrapping an adaptee
Decorator	`PRISM.H:123-157`	Three-layer struct embedding: `Box` → `BoxBM` → `BMOB`, each adding capabilities	Static struct composition, not dynamic wrapping; no common interface between layers
Decorator	`PRISM.H:306-333`	14-bit flag field composing 32 behavioural variants of interaction modes	Bitfield combinatorics replace the dynamic attachment of responsibilities
Behavioural			

Continued on next page

GoF Pattern	Evidence	DPaint Mechanism	Divergence from GoF
Command	`PAINTW.C:28-30`	Eight function-pointer slots encoding deferred interactive operations, swapped atomically by `IModeProcs()`	No command object; function pointers serve as first-class command tokens
Command	`MAINMAG.C:71`, `PRISM.H:396`	`didType` tag recording last operation for undo dispatch	Degenerate single-level command; type tag replaces command-object stack
Strategy	`PGRAPH.C:100-102`	`wrPixTable[]` maps paint modes to pixel-write functions; `paintProps[]` maps to Blitter configs	Parallel dispatch tables replace polymorphic strategy classes
Strategy	`GEOM.C`, `CONIC.C`	Geometric primitives accept a per-pixel function pointer parameter	Strategy passed as a function argument, not injected into an object
State	`PAINTW.C:237-271`	32-entry `IModeDesc` table with `nextIMode` chaining and `startProc` reconfiguration	Table-driven state machine; transitions swap function pointers, not object identity
Observer	`PANE.C`, `CCYCLE.C`	Pane callback registration with broadcast refresh; VBlank interrupt subscription for colour cycling	No subject/observer interface; Pane dispatch is spatial, VBlank is hardware-interrupt
Template Method	`PAINTW.C:166-209`	Fixed event-handling skeletons (`GoDown`, `GoUp`) calling pluggable function pointers	"Subclassing" via function-pointer swapping, not virtual methods

Continued on next page

GoF Pattern	Evidence	DPaint Mechanism	Divergence from GoF
Template Method	`MODES.C:121-158`	Pre-fabricated mode templates (`VTypeOps`, `XHTypeOps`, `CircTypeOps`) parameterised by drawing function	Higher-order functions replace abstract methods; templates are shared, not inherited
Creational			
Factory Method	`BITMAPS.C`, `DALLOC.C`	`AllocBitMap()`, `NewSizeBitMap()`, `DAlloc()` / `ChipAlloc()` with memory-floor guards	No abstract creator; factory functions are standalone, not overridden by subclasses
Factory Method	`CURBRUSH.C:141`	`SelPen()` dispatcher routing to `RoundPen`, `SquarePen`, `DotsPen` with shared finaliser	Parameterised dispatch replaces subclass-based factory instantiation
Singleton	`PRISM.C`	∼60 global variables constituting application-wide monostate; single instances of all display resources	Module-as-singleton via file-scope globals; no access-controlled instance method
Prototype	`BITMAPS.C:127-138`	`DupBitMap()` (shallow) and `CopyBitMap()` (deep clone via Blitter)	Blitter hardware serves as the copy engine; no `clone()` interface
Prototype	`MAINMAG.C:102-110`	Undo via triple-XOR Blitter swap between screen and `hidbm`	In-place prototype swap requiring zero additional memory

4.3.5. Patterns Not Observed

Several GoF patterns are notably absent in the design of *DPaint*, which we explain below:

- Creational design patterns:

  - *Abstract Factory*: all factories produce single concrete types without family grouping.
  - *Builder*: multi-step initialisation in `DPInit()` is monolithic procedural code.

- Structural design patterns:
  - *Bridge*: the hardware-specific code in *DPaint* is not separated from its abstraction through an independent interface hierarchy.
  - *Composite*: panes and menu items form flat linked lists, not recursive trees.
  - *Flyweight* is absent despite the memory-constrained environment; each graphical object is a distinct instance with no shared intrinsic-state pool.
  - *Proxy*: all objects are accessed directly without indirection or access control wrappers.

- Behavioural design patterns:
  - *Chain of Responsibility*: the pane hit-testing is spatial lookup, not voluntary request handling.
  - *Interpreter*: there is no domain-specific language or grammar parsing.
  - *Iterator* is absent as an abstraction; all iteration uses explicit `for`/`while` loops over linked lists (e.g., Pane lists in `PANE.C`, menu item lists in `MENU.C`), with no abstraction layer providing a generic iteration interface.
  - *Mediator*: panes do not communicate via a central coordinator.
  - *Memento* is absent in its GoF form: the undo mechanism uses bitmap copying (Prototype) with a type tag (Command), not opaque state-capture tokens.
  - *Visitor*: operations on the `BMOB` and `Pane` structures are implemented with direct function calls rather than double dispatch.

These absences are consistent with the platform constraints: tree structures consume pointer overhead ill-suited to 256 KB RAM; abstract interfaces

add indirection costs on a 7.09 MHz processor; and the single-user, single-document model eliminates the need for patterns that manage multiplicity (Flyweight, Composite, Iterator).

4.3.6. Summary

Table 7 provides a compact summary of all identified pattern antecedents. The analysis reveals that 11 of the 23 GoF design patterns have identifiable antecedents in *DPaint* source code, all implemented through C idioms—function pointers, struct embedding, dispatch tables, and bitfield composition—rather than the class hierarchies and virtual dispatch mechanisms that the GoF catalogue assumes. For every implemented pattern, the key structural divergence from the GoF specification is the same: the absence of polymorphism forces the pattern's intent to be realised through data-driven dispatch.

The seven additional architectural patterns (overlay system, two-buffer undo, per-object save-under, cooperative resource sharing, hardware register abstraction, graphics context stack, service locator) have no GoF analogues but are solutions to hardware constraints absent in modern software development, but still present with IoT devices.

The analysis identifies antecedents to 11 of the 23 GoF design patterns, all implemented through C idioms rather than class hierarchies. The seven additional architectural patterns have no GoF analogues but address hardware constraints specific to the Amiga platform.

> These findings suggest that design patterns are emergent properties of well-structured software facing recurrent design problems, independent of programming paradigm or knowledge of pattern catalogues. The engineering pressure of building a complex interactive system on severely constrained hardware led *DPaint*'s developers to some of the solutions that the GoF would formalise nine years later.

Beyond the GoF patterns identified above, the source code contains platform-specific idioms that have no formal catalogue counterpart.

4.4. RQ4: What coding idioms are present in the source code of DPaint?

As defined in Section 4.3, *patterns* have GoF analogues, while *idioms* are platform-specific solutions without GoF counterparts.

Several implementation idioms observed in *DPaint* have no direct GoF analogue but represent architectural solutions driven by the target platform's constraints.

Table 7: Summary of identified design pattern antecedents. Each entry lists the pattern family, specific pattern, and primary evidence location(s).

Family	**Pattern**	**Primary Evidence**
Behavioural	Command (8-slot procs)	PAINTW.C: 28–30
	Command (Undo tags)	MAINMAG.C: 71, PRISM.H: 396
	Command (Menu dispatch)	MENU.C: 159, 391
	Observer (Pane callbacks)	PANE.C, CCYCLE.C
	State (IMode FSM)	PAINTW.C: 237–271
	Strategy (Pixel writers)	PGRAPH.C: 100–102
	Strategy (Geo. primitives)	GEOM.C, CONIC.C
	Template Method	PAINTW.C: 166–209, MODES.C
Creational	Factory (Bitmap/Memory)	BITMAPS.C, DALLOC.C
	Factory (Pen dispatch)	CURBRUSH.C: 141
	Prototype (Bitmap clone)	BITMAPS.C: 127–138
	Prototype (Undo swap)	MAINMAG.C: 102–110
	Singleton/Monostate	PRISM.C
Structural	Adapter (Coordinates)	PRISM.H: 110–115
	Adapter (IFF Format)	DPIFF.C, ILBMR/W.C
	Decorator (BMOB layers)	PRISM.H: 123–157
	Decorator (IMode flags)	PRISM.H: 306–333
	Façade (Graphics)	PGRAPH.C
	Façade (Blitter)	BLITOPS.C, MASKBLIT.C
	Façade (Bitmap Mem.)	BITMAPS.C
	Façade (Windowing)	PANE.C
Architectural	Code Overlay System	PRISM.txt, PRISM.H: 409
	Cooperative Resource Sharing	PRISM.C: 343, DPIO.C: 42
	Graphics Context Stack	PGRAPH.C: 38–52
	Hardware Register Abstraction	PRISM.H: 382–390
	Inter-process Service Locator	HOOK.C, DPHOOK.H
	Per-object Save-under	BMOB.C: 311–403
	Two-buffer Undo	MAINMAG.C: 58–110

Before describing the individual idioms, we present quantitative indicators that characterise the coding practices in the source code.

4.4.1. Code Density and Documentation

The mean code density (executable lines divided by total lines) across the 88 files is 0.24, indicating that approximately one quarter of each file consists of executable statements. The mean comment density (comment lines divided by total lines) is 0.20. The highest comment density is in `DPHOOK.H` (0.76), a public interface file intended for use by external applications. The lowest comment densities (below 0.10) appear in `DPIO.C`, `MAINMAG.C`, and `TEXT.C` that have well-structured but sparsely documented procedural code.

4.4.2. Function Signatures

The 546 functions have a mean parameter count of 2.7 (median 2). The distribution is concentrated at the low end: 21% of functions take no parameters, 39% take one or two, 29% take three to five, and 11% take six or more. The five functions with the highest parameter counts are `PaletteTool` (25), `UndoSave` (21), `IModeProcs` (18), `NewIMode` (17), and `DPInit` (15). The high counts reflect the Understand SciTools metric `CountInput`, which counts all inputs to a function, including global variables read. These functions are coordinators that access many global state variables (display resources, bitmaps, palette data) to configure multiple subsystems simultaneously.

4.4.3. Preprocessor Usage

The mean preprocessor directive density (preprocessor lines divided by total lines) is 0.09. Header files exhibit the highest densities: `INTUALL.H` (0.88), `PRISM.H` (0.47), and `SYSTEM.H` (0.43). Among implementation files, preprocessor usage is low (typically below 0.05), confirming that macro-based inline expansion was used selectively rather than pervasively.

4.4.4. Coupling and Instability

Using Martin's instability index $I = Ce/(Ce + Ca)$, where Ca is afferent coupling (incoming dependencies) and Ce is efferent coupling (outgoing dependencies), the most stable files ($I = 0$) are the header files `SYSTEM.H` (Ca=58), `PRISM.H` (Ca=48), and `PNTS.H` (Ca=12), which are depended upon by many files but depend on nothing internally. The most unstable files ($I = 1$) are leaf files such as `PALBMS.C`, `MAGBITS.C`, `CLIP.C`, and `MASKBLIT.C`,

which depend on other files but are not depended upon themselves. This distribution is characteristic of a layered architecture where stable foundation files provide services consumed by volatile application-level files.

4.4.5. Architectural Idioms

The following seven idioms were identified through manual inspection of the source code. Each represents a platform-specific solution to the constraints of the Amiga 1000 hardware.

Code Overlay System. "Overlays are intended primarily for large applications that must run in a small amount of memory and leave more room for data. *Overlays* are groups of functions (and the data they require) that reside in memory only while in use. When the group in memory is no longer needed, the *overlay manager* reclaims the memory used by this group and loads in the next group of functions (and data)." [34].

DPaint has one root, 6 overlays of Level 1, and 2 overlays of Level 2:

0. Startup code, main program, menus, mouse and keyboard, toolbar and its icons, magnifier, etc.
    (a) Initialisation of *DPaint*
    (b) Usual drawing controls
    (c) Managing rotations
    (d) Handling colour palette
    (e) Performing symmetries
    (f) Reading/saving IFF files
        i. Reading, decompressing files
        ii. Compressing, writing files, adding icons

The Level 2 overlays pertain to reading or writing IFF files and are under the overlay of Level 1 pertaining to I/O, requesters, and general IFF functions. The linker control file `PRISM.txt` partitions the binary into a resident ROOT segment (38 modules: core event loop, UI, drawing primitives, magnification, Blitter operations) and Overlay segments (43 modules: initialisation, complex tools, transformations, palette editor, file I/O).

The `OVSInfo` struct (`PRISM.H`, lines 409–414) provides lifecycle callbacks: `sleepCursor` (displays a wait cursor during loading), `wakeCursor` (restores

the cursor), and `panic` (frees the brush bitmap when memory is critical). Three strategies are supported: `OVS_DUMB` (minimal memory, swaps on every cross-segment call), `OVS_SMART` (caching), and `OVS_LOAD_ALL` (preloads all overlays on 512 KB machines). Using overlays is essentially handling virtual memory manually, driven by the 256 KB RAM limitation.

The empty functions `InitRead()` and `InitWrite()` in files `INITREAD.C` and `INITWRIT.C`, respectively, force overlay loading from the program disk for disk operations before the user inserts their data disk to manage the single-floppy-drive constraint.

Cooperative Resource Sharing. `TmpRas`—a plane-sized Chip RAM buffer for flood fill, mask generation, and compositing—is strategically freed before operations that require Intuition to allocate save-under areas (menus: `PRISM.C`, line 343; file requesters: `DPIO.C`, line 42) and reallocated afterwards. This pattern of active memory-portfolio management has no modern analogue; it arises from the shared Chip RAM pool between the application and the operating system.

Graphics Context Stack. `PushGrDest`/`PopGrDest` (`PGRAPH.C`, lines 38–52) manage a four-deep stack of (`RastPort*`, clip `Box`) pairs, enabling nested rendering-target switches analogous to PostScript's `gsave`/`grestore`. This is an early instance of rendering context management: a four-deep rendering context stack implemented seven years before OpenGL 1.0 (1992) standardised `glPushMatrix`/`glPopMatrix`.

Hardware Register Abstraction. The `BlitterRegs` struct (`PRISM.H`, lines 382–390) overlays C field names onto memory-mapped I/O registers at address `$DFF040`, with `ioskip` fields accounting for unmapped register gaps. Symbolic Blitter Minterm constants (`REPOP`, `COOKIEOP`, `XOROP`) and the `BlitSize()` macro form a domain-specific vocabulary for Blitter programming.

Inter-process Service Locator. `SetHook()` (`HOOK.C`, line 14) creates a named Amiga OS message port wrapping a local `Hook` struct (containing a `MsgPort` and a data pointer). The companion `DPHOOK.H` defines a separate `DPHook` struct exposing the application's bitmap, viewport, and format, which is published through the hook. External applications locate *DPaint* via the Amiga OS function `FindPort("DeluxePaint")`, which returns the named message port created by `SetHook()`. The *DPaint*-internal `FindHook()` (defined in `HOOK.C`) wraps this `FindPort()` call. When multiple *DPaint* instances run, `FindHook()` in `DPINIT.C` detects the existing port, and the new

instance shares the original's screen mode and bitmap via the `DPHook` structure's `refCount` field. This constitutes a proto-service-registry pattern with reference counting.

Per-object Save-under. Each `BMOB` carries a `save` buffer. `ShowBMOB()` (`BMOB.C`, line 311) captures the background before painting; `ClearBMOB()` (line 327) restores it. `ChangeBMOB()` (line 350) optimises moves by computing the rectangular complement (`BoxNot()`) of old and new positions, minimising Blitter transfers. The `TOOBIGTOPAINT` flag enables graceful degradation to an XOR outline when memory is insufficient for the save buffer.

Two-buffer Undo Architecture. The screen bitmap serves as the live drawing surface; `hidbm` stores the last committed state. Drawing is visible immediately on screen. `UndoSave()` (`MAINMAG.C`, line 87) copies the changed region (tracked via the bounding box `chgB`) from screen to `hidbm`, making changes permanent. `Undo()` swaps the two via the triple-XOR Blitter trick. The `Painting` flag (`MAGWIN.C`, line 26) partitions rendering into two tracks: XOR feedback (tracked in `tmpChgBox`, erased automatically) and permanent paint (tracked in `chgB`, committed on the next action).

The seven architectural idioms identified in *DPaint* have no direct GoF analogues but represent practical solutions to the constraints of the Amiga 1000 platform. The code overlay system, cooperative resource sharing, and hardware register abstraction address the 256 KB memory ceiling; the two-buffer undo and per-object save-under optimise interactive performance on a 7.09 MHz processor. These idioms demonstrate that platform constraints can drive architectural innovation.

Having characterised the architectural patterns and platform-specific idioms, we now study the complexity of the source code.

4.5. RQ5: How was complexity distributed across the different components of DPaint, and why?

To understand how complexity is distributed in *DPaint* source code, we analysed cyclomatic complexity at both file and function levels, grouped files by functional category, and examined the relationships between complexity, code size, and nesting depth.

Table 8: Top 10 files by sum cyclomatic complexity. Avg CC is the mean cyclomatic complexity per function (Sum CC divided by the number of functions).

File	Sum CC	LOC	Max CC	Avg CC	Max Nest
PALETTE.C	117	1,337	23	3.8	3
CHPROC.C	95	253	74	8.6	3
MODES.C	93	457	6	1.3	1
PANE.C	62	417	13	2.7	4
MAINMAG.C	58	451	6	1.8	2
MENU.C	53	454	22	3.3	2
PAINTW.C	47	429	5	2.0	2
DPIO.C	41	352	6	2.7	4
BMOB.C	37	417	11	2.5	2
PGRAPH.C	37	313	6	1.6	3

4.5.1. File-Level Complexity Distribution

Of the 88 source files analysed by Understand SciTools (77 `.C` and 11 `.H` files, excluding the linker control file `PRISM.txt`), 56 have non-zero cyclomatic complexity. The remaining 32 are header files or files without function definitions. The total cyclomatic complexity across all files is 1,223, with a minimum of 1, an average of 21.8, and a maximum of 117 (`PALETTE.C`).

The distribution is highly skewed: six files account for 39% of total complexity, while 14 of the 56 files with non-zero complexity have a sum complexity below 5. Table 8 lists the ten most complex files.

`PALETTE.C` dominates the complexity ranking because it implements the full colour palette editor, including RGB and HSV sliders, colour cycling range management, colour spreading, and the copy/exchange operations—all within a single file that also manages its own floating window. Despite its high sum complexity (117), its average per-function complexity is only 3.8, indicating that the complexity arises from the *number* of operations rather than from individually convoluted functions.

`CHPROC.C` presents the opposite profile: its sum complexity (95) derives almost entirely from a single function, `mainCproc` (CC=74), which contains a long `switch` statement dispatching keyboard shortcuts to their corresponding actions. The remaining 10 functions in the file average CC=2.1.

`MODES.C` has the third-highest sum complexity (93) but a low average (1.3 per function). Its 74 functions are predominantly one-line mode-initialisation

wrappers, each calling a template function with specific parameters.

4.5.2. Function-Level Complexity Distribution

At the function level, 546 functions have a mean cyclomatic complexity of 2.2 and a median of 1, with the distribution sharply concentrated at the low end:

- **CC = 1** (trivial): 314 functions (57.5%)
- **CC 2–5** (simple): 200 functions (36.6%)
- **CC 6–10** (moderate): 21 functions (3.8%)
- **CC 11–20** (complex): 8 functions (1.5%)
- **CC > 20** (very complex): 3 functions (0.5%)

Over 94% of functions have a cyclomatic complexity of 5 or below, indicating that the source code overwhelmingly comprises short, single-purpose functions. The three functions with CC above 20 are `mainCproc` (CC=74, keyboard dispatch), `PaletteTool` (CC=23, palette editor event handler), and `DoBrs` (CC=22, brush operation dispatcher). All three are event-dispatching functions whose complexity arises from the number of user-interface cases they handle, not from algorithmic intricacy.

4.5.3. Complexity by Functional Category

To examine whether certain types of functionality consistently produce higher complexity, we grouped the 77 source files into seven functional categories based on their primary purpose. Table 9 summarises the results.

The UI and interaction category has both the highest total complexity (472, 38.6% of the total) and the highest average per file (29.5). This is expected: interactive event handling requires branching on user input (mouse position, key presses, menu selections, mode states), producing high cyclomatic complexity even when each branch is individually simple.

The palette and colour category ranks second (208, 17.0%), driven primarily by `PALETTE.C`. Colour management involves multi-dimensional computations (RGB $\leftrightarrow$ HSV conversion, colour spreading algorithms, cycling range logic) that inherently require more decision paths.

By contrast, the file I/O category has the lowest average complexity (4.5 per file). The IFF read/write modules (`IFFR.C`, `IFFW.C`, `ILBMR.C`, `ILBMW.C`)

Table 9: Complexity distribution by functional category.

Category	Sum CC	LOC	Files	Avg CC/File
UI / Interaction	472	4,389	16	29.5
Palette / Colour	208	2,299	8	26.0
Bitmap / Blitter	161	3,089	13	12.4
Transforms	131	1,206	7	18.7
System / Init	110	1,557	15	7.3
Drawing / Graphics	96	1,284	8	12.0
File I/O	45	2,058	10	4.5

follow a linear, sequential structure dictated by the IFF file format specification, which leaves little room for branching.

4.5.4. Nesting Depth

The maximum nesting depth across all files is 4, observed in five files: `CCYCLE.C`, `DPIO.C`, `HSV.C`, `PANE.C`, and `ROT90.C`. The distribution is concentrated at the shallow end: 43 files have maximum nesting 0 (no nested control structures), 19 have depth 1, 14 have depth 2, and 7 have depth 3. The low nesting depths reflect the flat, linear control flow that dominates the source code and contributes to its readability.

The five files reaching depth 4 share a common pattern: they handle either user events and error conditions (`DPIO.C`, `PANE.C`) or perform multi-step numerical computations with boundary checks (`HSV.C`, `ROT90.C`, `CCYCLE.C`).

4.5.5. Correlation Between Size and Complexity

File size (LOC) and sum cyclomatic complexity are positively correlated (Pearson $r = 0.72$, over the 56 files with non-zero complexity), indicating that larger files tend to be more complex. However, the relationship is not purely linear: `CHPROC.C` (253 LOC, CC=95) achieves high complexity in a small file through a single dense switch statement, while `FNREQ.C` (795 LOC, CC=25) is large but structurally simple because it consists primarily of data declarations for the file requester UI.

This correlation suggests that complexity in *DPaint* is driven by the *scope of responsibility* of a file rather than by poor engineering: files that handle more user-facing operations (palette editing, mode switching, event dispatching) accumulate more decision paths, while utility files that serve a single well-defined purpose remain simple regardless of their size.

Complexity in *DPaint* is concentrated in a small number of UI-facing files that handle event dispatching and user interaction, while the majority of the source code (94% of functions at CC $\leq$ 5) consists of simple, single-purpose routines. The most complex file (`PALETTE.C`, CC=117) and the most complex function (`mainCproc`, CC=74) both derive their complexity from the breadth of user-interface cases they handle, not from algorithmic intricacy. The low median function complexity (1), shallow nesting depths (maximum 4), and the concentration of complexity in identifiable "hot spots" indicate a well-managed source code where complexity is localised rather than diffused, a characteristic of disciplined engineering even in the absence of modern complexity-management tools.

The results above reveal a source code that combines low average complexity with deliberate architectural organisation. We now discuss the implications of these findings.

5. Discussions

We now summarise our results and answers and expand these answers into lessons learned and lessons for the future. We also discuss threats to the validity of our results and discussions.

5.1. Summary

In this article, we answered five research questions. We first asked how and by whom *DPaint* was developed, and what its overall quality was (RQ1). *DPaint* was developed primarily by Dan Silva with five other contributors, and its 89-file, 17,001-line source code compiles with only warnings on a modern SAS/C compiler, indicating high code quality for its era. We then examined its architectural styles (RQ2) and found a hub-and-spoke architecture centred on `PRISM.C` and `PRISM.H`, with five communities identified through greedy modularity optimisation. Turning to its design (RQ3), we identified antecedents to 11 of the 23 GoF patterns, all implemented through C-language constructs—function pointers, struct embedding, and dispatch tables—rather than class hierarchies. We also looked for platform-specific coding idioms (RQ4) and found seven, including a code overlay system, cooperative resource sharing, and hardware register abstraction, all driven by the 256 KB memory constraint. Finally, we analysed how complexity is

distributed across components (RQ5): it concentrates in UI-facing event-dispatching files, and 94% of the 546 functions have a cyclomatic complexity at or below 5.

5.2. Lessons Learned

Five key lessons emerge from our analysis. First, design patterns are not artefacts of the object-oriented paradigm. *DPaint*'s developers independently arrived at structural solutions that the GoF would formalise nine years later, using function pointers, struct embedding, and dispatch tables in place of classes and virtual methods. This suggests that patterns arise from recurring design problems, not from knowledge of pattern catalogues.

Second, hardware constraints can drive quality architecture. The 256 KB memory ceiling forced *DPaint*'s developers to create modular, low-coupling designs where each file has a clear responsibility. The hub-and-spoke architecture centred on `PRISM.C/H` emerged not from architectural planning but from the practical need to share scarce resources (Chip RAM, screen bitmaps, Blitter hardware) without duplicating state.

Third, every abstraction must justify its memory footprint. We argue that the absence of certain design patterns, like Composite, Iterator, and Flyweight, is not accidental: tree structures consume pointer overhead, iterator abstractions add indirection costs, and shared intrinsic-state pools require management overhead, all ill-suited to 256 KB RAM.

Fourth, a single-developer project can achieve a high degree of consistency. Dan Silva's 17,001 lines of code exhibit uniform naming conventions (8+3 uppercase), consistent use of pre-ANSI C idioms, and a coherent architectural vision. The low average cyclomatic complexity (2.2 per function) and the concentration of complexity in a few well-identified files (`PALETTE.C`, `CHPROC.C`) suggest disciplined engineering.

Fifth, the absence of modern tools does not preclude quality. Without version control, IDEs, linters, or automated testing, *DPaint* source code compiles with only warnings on a modern compiler and runs correctly on original hardware. The code's quality is evidenced by its commercial success and its continued compilability 40 years later.

5.3. Lessons for the Future

The techniques observed in *DPaint* remain directly relevant to modern constrained-environment development. The code overlay system (partitioning a binary into resident and demand-loaded segments) is functionally equiv-

alent to dynamic library loading, plugin architectures, and microservice decomposition, all of which manage the same fundamental trade-off between memory footprint and functionality.

The cooperative resource sharing pattern, where `TmpRas` (the temporary raster buffer Intuition borrows for off-screen rendering) is freed before Intuition allocations and reallocated afterwards, anticipates modern resource pooling strategies in IoT devices, where GPU memory, network buffers, and file handles must be shared among subsystems.

The hardware register abstraction, overlaying C struct field names onto memory-mapped I/O registers, remains the standard approach in embedded systems programming. Modern Hardware Abstraction Layers (HALs) in CMSIS, Zephyr, and Linux device drivers use the same technique, validating a design decision made in 1985.

For computer science education, *DPaint* offers a compact, self-contained case study in system programming. Its 17,001 lines are small enough for a student to read in full, yet complex enough to exhibit real architectural, design, and coding decisions. The source code shows that constraints breed creativity, a lesson directly applicable as IoT devices, edge computing, and battery-powered systems impose memory and power budgets reminiscent of the Commodore Amiga era.

5.4. Threats to Validity

Before drawing final conclusions, we consider the limitations of our study. Although we analysed the source code of Deluxe Paint thoroughly and from different angles, it is necessary to acknowledge the potential limitations and threats to the validity of our findings.

First, our study focuses on one specific version of the source code, and different versions or revisions certainly exhibit variations in design, implementation, or optimisation techniques. Future research could explore the evolution of the source code over time to gain a more complete understanding of its development history.

Second, our analysis is necessarily limited by the available documentation and comments within the source files. While the code is generally well-commented, there may be undocumented features, edge cases, or historical context that are not apparent in the source code alone. Interviews with the original developers or access to design documents from the time of the development would provide additional insights and help refine our findings.

Third, while we selected a set of key files and subsystems for detailed analysis, there may be other aspects of the source code that are not covered in this study. A more exhaustive examination of all the source files could potentially reveal additional patterns, techniques, or areas of interest. The selection of these files is based on the centrality and community detection analyses described in Section 3.2: we prioritised files with the highest betweenness centrality, in-degree, and cross-community connectivity for detailed inspection.

Finally, we recognise that our interpretations and conclusions are shaped by our own knowledge, experiences, and biases as researchers. Although we tried to maintain an objective and impartial perspective, other researchers might derive different insights or emphasise different aspects of the source code based on their own backgrounds and expertise. In particular, the design pattern identification (RQ3) was performed by the authors without independent validation by a separate team. To mitigate this threat, we provide a replication tool (`verify_patterns.py`) that mechanically confirms the presence of every cited code structure, enabling reviewers and future researchers to verify our claims independently.

Despite these limitations, we believe that our analysis provides insights into the design and implementation of Deluxe Paint and that the lessons learned from this landmark piece of software engineering can inform modern practitioners.

6. Conclusion

In this article, we presented a detailed analysis of the source code of *Deluxe Paint* (*DPaint*), a seminal graphics program developed for the Commodore Amiga platform in 1985 with the goal **to understand the constraints that guided the architecture, design, and implementation of *DPaint* to handle the computational and memory constraints of the Amiga 1000** (7.09 MHz CPU, with 256 KB of RAM by default).

Through a combination of quantitative metrics computations, dependency analyses, and qualitative code examination, we identified architectural styles, design patterns, coding idioms, as well as optimisation techniques, that enabled *DPaint* to deliver a quality and efficient user experience on the constrained hardware of the era.

Our analysis of the file-level and function-level metrics of the C code revealed code focused on modularity, low complexity, and efficiency. The

examination of key files revealed developers' mastery of the Amiga custom hardware and their ability to create abstractions that enable high performance while maintaining readability.

The exploration of critical subsystems, such as colour palette management (`PALETTE.C`), initialisation (`DPINIT.C`), file I/O (`DPIO.C`), magnification (`MAINMAG.C`), and text rendering (`TEXT.C`), showed the technical skill of the *DPaint* developers. Their use of techniques, e.g., double buffering, caching, incremental updates, and tight integration with the Amiga hardware architecture, allowed them to create feature-rich software that pushed the boundaries of what was possible on the platform. We also showed that components and patterns were already present to abstract from the hardware, avoid code duplication, and make the implementation more efficient.

We used this understanding to draw lessons and parallels with modern software development, in particular on constrained devices, such as IoT devices. We concluded that the techniques used 40 years ago still apply today, and their "rediscovery" could benefit current development.

These techniques find direct parallels in modern embedded practice: code overlays correspond to firmware partitioning in over-the-air (OTA) update systems with A/B slots (i.e., two firmware partitions, one active while the other receives the update), cooperative resource sharing mirrors memory pool management in real-time operating systems such as FreeRTOS, and hardware register abstraction remains the foundation of vendor-provided HAL libraries (e.g., ARM CMSIS).

Moreover, several *DPaint* interface concepts persist in modern graphics software: the floating colour palette (adopted by Adobe Photoshop and GIMP), the custom brush (now "stamp" or "clone" tools in most editors), the symmetry drawing mode (present in Procreate and Krita), and the spare page (a precursor to modern layer systems).

Future work could reproduce this analysis for other landmark Amiga software, such as Directory Opus (file manager), ProTracker (music tracker), or the Video Toaster (video editing suite), to determine whether similar patterns and idioms emerge across different application domains on the same constrained platform. In addition, a module-level instability analysis, aggregating the coupling metrics of files grouped by the naming-prefix modules is a natural extension of this work.

7. Acknowledgments

Giuseppe Destefanis would like to dedicate this article to Prof. Gennaro Moretta, his Italian literature professor in junior high school (in Sedini - Sardinia, Italy), who, in the first half of the 1990s, guided him through the world of the Amiga computer, sharing his knowledge and passion. It was through his introductory computer course that Giuseppe first discovered his passion for computer science. Prof. Moretta's influence extended beyond the classroom. His imprinting is fundamental to who Giuseppe is today, both as a person and an academic. Without Prof. Moretta's guidance, encouragement, and generosity, Giuseppe might never have discovered his passion or had the opportunity to pursue it.

References

[1] J. Maher, Deluxe Paint, MIT Press, 2012.

[2] J. Maher, The future was here: the Commodore Amiga, MIT Press, 2012.

[3] F. Alizadeh, A. Mniestri, A. Uhde, G. Stevens, On appropriation and nostalgic reminiscence of technology, in: Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems, CHI EA '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1–6. doi:10.1145/3491101.3519676.
URL https://doi.org/10.1145/3491101.3519676

[4] S. , R. Cheston, G. Christopher, J. Meyrick, Nostalgia as a psychological resource for people with dementia: A systematic review and meta-analysis of evidence of effectiveness from experimental studies, Dementia 19 (05 2018). doi:10.1177/1471301218774909.

[5] P. Galloway, Retrocomputing, archival research, and digital heritage preservation: A computer museum and ischool collaboration, Library Trends 59 (2011) 623–636. doi:10.1353/lib.2011.0014.

[6] H. Tomari, K. Hiraki, Keeping old computers alive for deeper understanding of computer architecture, in: Proceedings of the Workshop on Computer Architecture Education, WCAE '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 1–7. doi:

10.1145/2795122.2795127.
URL https://doi.org/10.1145/2795122.2795127

[7] Z. Zhang, Retrocomputing in contemporary integrative stem education, in: 2023 IEEE Integrated STEM Education Conference (ISEC), 2023, pp. 338–342. doi:10.1109/ISEC57711.2023.10402127.

[8] A. Reinhard, Archaeogaming – An Introduction to Archaeology in and of Video Games, Berghahn Books, 2018.

[9] B. Books, S. Dyer, Sinclair ZX Spectrum: A Visual Compendium, Bitmap Books, 2015.

[10] B. Books, S. Dyer, Commodore Amiga: A Visual Compendium, Bitmap Books, 2015.

[11] M. Barton, Dungeons and Desktops: The History of Computer Role-Playing Games, A K Peters/CRC Press, 2008.

[12] J. Thomas, B. David, Landscape archaeology: introduction, Left Coast Press, 2008, Ch. 1, pp. 27–43.

[13] J. Aycock, Amnesia Remembered: Reverse Engineering a Digital Artifact, Digital Archaeology: Documenting the Anthropocene, Berghahn Books, 2023.
URL https://www.berghahnbooks.com/title/AycockAmnesia

[14] Y. Rochat, A quantitative study of historical video games (1981–2015), Historia Ludens: The Playing Historian (2019). doi:10.4324/9780429345616-1.
URL https://infoscience.epfl.ch/handle/20.500.14299/164377

[15] J. Lawler, S. Smith, Reprogramming the history of video games: A historian's approach to video games and their history, International Public History 4 (1) (2021) 47–54 [cited 2026-03-01]. doi:doi:10.1515/iph-2021-2018.
URL https://doi.org/10.1515/iph-2021-2018

[16] C. Rollinger, Classical Antiquity in Video Games: Playing with the Ancient World, IMAGINES ? Classical Receptions in the Visual and Performing Arts, Bloomsbury Publishing, London :, 2020.
URL https://digital.casalini.it/9781350066649

[17] B. Ager, Classical antiquity in video games: Playing with the ancient world (a bryn mawr classical review), Online, last accessed on 26/03/01 at https://bmcr.brynmawr.edu/2021/2021.03.02/ (March 2021).

[18] D. Belyaev, U. Belyaeva, Historical video games in the context of public history: Strategies for reconstruction, deconstruction and politization of history, Galactica Media: Journal of Media Studies 4 (1) (2022) 51–70. doi:10.46539/gmd.v4i1.204.
URL https://galacticamedia.com/index.php/gmd/article/view/204

[19] D. Spring, Gaming history: computer and video games as historical scholarship, Rethinking History 19 (2) (2015) 207–221. doi:10.1080/13642529.2014.973714.
URL https://doi.org/10.1080/13642529.2014.973714

[20] J. Aycock, Retrogame Archeology: Exploring Old Computer Games, Springer, 2016.
URL https://api.semanticscholar.org/CorpusID:195941640

[21] B. Dym, E. Simpson, O. Fong, libi striegl, The internet is not forever: Challenges and sustainability in video game archiving and preservation, Journal of Electronic Gaming and Esports 1 (1) (2023) jege.2022–0041. doi:10.1123/jege.2022-0041.
URL https://journals.humankinetics.com/view/journals/jege/1/1/article-jege.2022-0041.xml

[22] O. Kuhnke, By shutting down eshops, nintendo again stands in the way of video games' legacy, Online, last accessed on 26/03/01 at https://www.gamespot.com/articles/by-shutting-down-eshops-nintendo-again-stands-in-the-way-of-video-games-legacy/1100-6501634/ (March 2022).

[23] D. Pigott, Online historical encyclopaedia of programming languages, Online, last accessed on 26/03/01 at http://hopl.info/why.html (1995).

[24] S. Monnier, M. Sperber, Evolution of emacs lisp, Proc. ACM Program. Lang. 4 (HOPL) (Jun. 2020). doi:10.1145/3386324.
URL https://doi.org/10.1145/3386324

[25] A. A. Hagberg, D. A. Schult, P. J. Swart, Exploring network structure, dynamics, and function using networkx, Python in Science Conference (2008). doi:10.25080/TCWV9851.
URL https://doi.org/10.25080/TCWV9851

[26] L. C. Freeman, A set of measures of centrality based on betweenness, Sociometry 40 (1) (1977) 35–41.
URL http://www.jstor.org/stable/3033543

[27] T. M. J. Fruchterman, E. M. Reingold, Graph drawing by force-directed placement, Software: Practice and Experience 21 (11) (1991) 1129–1164. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380211102, doi:https://doi.org/10.1002/spe.4380211102.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380211102

[28] A. Clauset, M. E. J. Newman, C. Moore, Finding community structure in very large networks, Phys. Rev. E 70 (2004) 066111. doi:10.1103/PhysRevE.70.066111.
URL https://link.aps.org/doi/10.1103/PhysRevE.70.066111

[29] R. Albert, H. Jeong, A.-L. Barabási, Error and attack tolerance of complex networks, Nature 406 (6794) (2000) 378–382. doi:10.1038/35019019.
URL http://dx.doi.org/10.1038/35019019

[30] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.

[31] R. C. Martin, Agile software development: principles, patterns, and practices, Prentice Hall PTR, 2003.
URL http://dl.acm.org/citation.cfm?id=515230

[32] T. McCabe, A complexity measure, IEEE Transactions on Software Engineering SE-2 (4) (1976) 308–320. doi:10.1109/TSE.1976.233837.

[33] J. Morrison, "ea iff 85" standard for interchange format files, Online, last accessed on 26/03/01 at https://wiki.amigaos.net/wiki/EA_IFF_85_Standard_for_Interchange_Format_Files (1985).

[34] S. I. Inc., SAS/C Development System User's Guide, Volume 1: Introduction, Compiler, Editor, SAS Institute Inc., 1993, version 6.50.