Implementation of hierarchical scaleable aggregation in an object-oriented image processing environment

Daniel Y. Sandholdt

Recommended Citation
https://scholarworks.umt.edu/etd/5076
The University of MONTANA

Permission is granted by the author to reproduce this material in its entirety, provided that this material is used for scholarly purposes and is properly cited in published works and reports.

** Please check "Yes" or "No" and provide signature **

Yes, I grant permission

No, I do not grant permission

Author's Signature

Date 12/6/99

Any copying for commercial purposes or financial gain may be undertaken only with the author's explicit consent.
Implementation of Hierarchical Scaleable Aggregation in an Object-Oriented Image Processing Environment

by

Daniel Y. Sandholdt

B.S. The University of Montana, 1996

presented in partial fulfillment of the requirements

for the degree of

Master of Science

The University of Montana

1999

Approved by:

[Signature]

Chairperson

[Signature]

Dean, Graduate School

12-16-99

Date
Image segmentation is the process of partitioning an image into a set of non-overlapping regions. However, many segmentation methods do not guarantee image partitioning resolution consistency. A method that does is called Hierarchically Scaleable Aggregation. This paper presents the development of an implementation of Hierarchically Scaleable Aggregation in the context of Bohem’s “Spiral Model of Software Development.” In its final form, the Hierarchically Scaleable Aggregation implementation is transformed from a solitary application to being a component in a graphical user interface based image processing environment.
Table of Contents

Abstract .................................................................................................................................................. ii
Table of Contents ................................................................................................................................. iii
Index of Tables ...................................................................................................................................... iii
Index of Figures .................................................................................................................................. iii
1. Introduction ......................................................................................................................................... 1
2. Development Model .......................................................................................................................... 4
  2.1 Spiral Previews ............................................................................................................................. 8
  2.2 Spiral Model Table Representation ............................................................................................. 9
  2.3 Spiral 1: Prior Development ....................................................................................................... 10
  2.4 Spiral 2: First Implementation ................................................................................................... 13
  2.5 Spiral 3: Canonical Ordering ..................................................................................................... 16
  2.6 Spiral 4: Prototype GUI ............................................................................................................. 20
  2.6 Spiral 5: Beta Quality ................................................................................................................. 22
3. Specific design and Implementation Considerations ......................................................................... 25
  3.1 Property Files ............................................................................................................................ 26
  3.2 Basic Image Processing ............................................................................................................. 27
  3.3 File Naming Problem ................................................................................................................. 29
  3.4 Objects ....................................................................................................................................... 29
  3.5 Nodes .......................................................................................................................................... 30
4. Summary and Conclusions ................................................................................................................ 34
Appendix A.1: Definition and Description of Rule-Based Image Aggregation ...................................... 36
Appendix A.2: Advantages of Merge .................................................................................................. 39
Appendix B: Canonical Ordering Report .............................................................................................. 42
Appendix C: Prototype GUI Report ..................................................................................................... 46
  Appendix C.1: System Structure of Original Merge and First Prototype .......................................... 48
  Appendix C.2: Details of First Prototype Package Structure .............................................................. 67
  Appendix C.3: Details of First Prototype Exceptions ........................................................................ 69
  Appendix C.4: Details of First Prototype Option Package Standard Class Interface .......................... 70
  Appendix C.5: Details of First Prototype GUI .................................................................................. 72
Appendix D.1: Features of jMERGE (v0.826) ..................................................................................... 73
Appendix D.2: Interview with a BetaUser ........................................................................................... 76
Bibliography .......................................................................................................................................... 79

Index of Tables

Table 1: Spiral Model Table Representation Template ........................................................................ 9
Table 2: Spiral 1: Prior Development .................................................................................................. 11
Table 3: Spiral 2: First Implementation .............................................................................................. 15
Table 4: Spiral 3: Canonical Ordering ................................................................................................ 18
Table 5: Spiral 4: Prototype GUI ........................................................................................................ 20
Table 6: Spiral 5: Beta Quality ........................................................................................................... 23

Index of Figures

Figure 1: Spiral of Theoretical Project ............................................................................................... 5
Figure 2: Rapid Prototype Development Model ................................................................................ 25
1. Introduction

While numerous methods exist for performing basic image segmentation [Haralick85.2], special consideration must be given to the correct segmentation of images that represent a ground scene. As described by Woordcock and Harward, “...pixels are assumed to be representative samples of objects in the ground scene. When pixels are large relative to ground objects, individual pixels often cover parts of two or more objects, resulting in mixed pixels, and the effectiveness of analysis is undermined. Similarly, when the pixels become very small relative to objects, the internal variance of the objects adversely affects the results of the analysis. The ideal situation is reached when the elements of analysis in the image correspond to the objects in the ground scene. The objective of image segmentation is to partition the image into a set of regions which correspond to the objects in the ground scene and will serve as the basis for future analysis [Woordcock92].” Regardless of change in image resolution this partitioning should be consistent, that is no new region boundaries are introduced when shifting from a fine to coarser resolution. This property of segmentation is not guaranteed by many common image processing techniques [Ford98].

A segmentation method that does guarantee image partitioning resolution consistency is called Hierarchically Scaleable Aggregation. In
Ford et al. describe hierarchical scaleable aggregation as a "technique [that] can be applied to very large, high resolution imagery to produce multiple enhanced intermediate-resolution images. This set of intermediate images can in turn be used to produce depictable images at a variety of scales, yet avoid the most common types of depiction anomalies."

In private correspondence with Dr. Ford, he further explained that the process of Hierarchically Scaleable Aggregation has two key characteristics that set it apart from other more traditional filtering and reduction-via-clustering approaches. First, the classification process, which is numerically intensive, is done only once, producing (in near-real-time) a base-line classified image with a minimum unit size of 1. Next, multiple derived datasets can be produced from this base map in parallel, aggregated at different minimum unit thresholds. Each dataset is guaranteed to have a hierarchical scaling property based on the unit threshold. It is easiest to describe this property in the special case of 2-D datasets where the units in question are areas. The hierarchical scaling property guarantees that for $X > Y$, when the areas in a derived image with unit threshold $X$ are compared with areas in a derived image with threshold $Y$:

- every "border" in $X$ is present in $Y$, and
• some borders in Y are "erased" in X (i.e., borders are eliminated and pixels in certain areas re-classified to aggregate small areas into larger areas)

From such a description it should be possible to actually implement a working version of an image processing program that is based upon hierarchical scaleable aggregation. In fact, multiple implementations have been created in a variety of programming languages to suit the needs of a variety of applications. Whenever user requirements changed, a new implementation had to be written. There was no easy means of incorporating most application-specific user requests into a particular program, short of making drastic changes that represented an effort comparable to writing a completely new implementation. This points to the obvious need for an image processing environment that is flexible enough to easily accommodate new user requirements, and at the same time has a flexible underlying structure so that programmers can perform these modifications with ease, and not unknowingly disrupt previously available services. The following chapters will discuss the development of this type of image processing environment, with the hierarchically scaleable aggregation algorithm at its core.
2. Development Model

The term image processing refers to a range of computational techniques used to “improve” an image. What constitutes an improved image is relative to the post-processing goals of the user. In one application the user may prefer to have contrast improved or blurring removed, whereas in another the user may prefer to have noise and lens distortion removed. One image processing technique that is of wide interest is image segmentation, which attempts to partition an image into areas of similar properties. According to Haralick and Shapiro, “… regions of an image segmentation should be uniform and homogenous with respect to some characteristic such as gray tone or texture. Region interiors should be simple and without many small holes. Adjacent regions of a segmentation should have significantly different values with respect to the characteristic on which they are uniform. Boundaries of each segment should be simple, not ragged, and must be spatially accurate [Haralick85].”

Like any other software development project, the development of image processing software should follow standard industry practice. Small-scale software projects can simply be written using little in the way of development methodology, but projects that are “large” in size and/or in time need to be developed according to careful plans. A widely accepted, excellent method of developing software is called the
spiral model [Bohem88]. The spiral model is both a technique for prescribing or planning project activity, based on identifying and minimizing risks associated with the project and a pictorial means of describing a project's evolutionary history. A typical spiral model depiction of a hypothetical project may look like this [Bohem88].

![Figure 1: Spiral of Theoretical Project](image)

In the spiral model, the spiral itself represents the development of the project as it passes through various phases over time. Quadrants are rough groupings of related work, as described in more detail below. Key points in the spiral model are where the spiral crosses from one quadrant to another, indicating a shift in the type of activity being...
performed. It is also important to note that depending on the level of detail included by the modeler, each phase could be described in more detail, with its own development spirals.

Though it is possible to classify software development activities in a number of ways, the following four classes are typically used as the four quadrants in the development space.

- **Quadrant 1: Determine objectives, alternatives, and constraints.**
  
  This represents the collection of activities used to determine or refine project objectives, come up with alternative ways to reach those objectives, and identify various constraints that may be imposed on those alternatives. The goal of this section is first to record the objectives, alternatives, and constraints, and then to come up with a strategy to consider how the objectives might be achieved.

- **Quadrant 2: Evaluate alternatives, identify and resolve risks.**
  
  What sets Boehm’s spiral model apart from other software development models is that it is risk driven. At any given time it is important to identify and categorize the risks that are involved. This section is devoted to looking at alternatives and constraints in the context of the risks they pose to the project’s success. The designer moves the project forward by identifying various risks and determining which alternative reduces risk the
most. This is actually an especially difficult section because it is hard to correctly perform risk analysis effectively when only partial information is available regarding objectives, constraints, and costs. This is exactly why a typical project spirals through this quadrant several times. During each pass, the risk assessment is repeated in the context, as more information about that project becomes available. Prototypes are often developed as a way to assess risk (e.g. to determine feasibility).

- Quadrant 3: Develop and verify next-level product.

Once risks have been identified and resolved to the extent possible with the information on hand, this section encapsulates specific development activity. Software is designed, code is written, modules or subsystems are tested, or other implementation activity, appropriate to this stage of project development, is completed. Again, note that this quadrant will be repeated several times, reflecting implementation activity that develops in a non-linear fashion in accordance with other activities encapsulated within the other quadrants.

- Quadrant 4: Plan next phases.

Artifacts created in the previous three quadrants, whether they are designs, code, or other implementation activities, are considered in creating evaluations of the current state of the
project, potential new goals, or requirement modifications. A decision is then made about whether to continue the project. This evaluation becomes part of a set of new objectives to be considered when the spiral enters quadrant one on the next pass.

2.1 Spiral Previews

The development of a system of meaningful size can not be completed in one spiral iteration. The development of this project went through numerous spirals, many of which contained their own subspirals. “Spiral 1: Prior Development” is a summary of the original proofs of concepts and development of the first aggregation systems. “Spiral 2: First Implementation” covers the training of a new group of implementers with the development tool of choice being Java. “Spiral 3: Canonical Ordering” takes a step from implementation back to study whether the aggregation algorithms that have been implemented so far are actually optimal. “Spiral 4: Prototype GUI” covers the development of a basic graphical user interface with which to interface with a modified version of an existing aggregation implementation. The final spiral presented here, though not necessarily the final spiral, is “Spiral 5: Beta Quality”, which addresses the issues surrounding the creation of a professional quality implementation and processing environment.
2.2 Spiral Model Table Representation

A spiral diagram charts the general course of a project, but at times it can be used effectively to record the specific results of a project. The most common way to record this information is in tables,

Table 1: Spiral Model Table Representation Template

<table>
<thead>
<tr>
<th>Quadrant 1</th>
<th>Quadrant 2</th>
<th>Quadrant 3</th>
<th>Quadrant 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints: List of known constraints to attaining objectives.</td>
<td>Risk Resolution: List of actions to be taken to circumvent identified risks.</td>
<td></td>
<td>Decision: List of the anticipated goals of the next spiral iteration.</td>
</tr>
</tbody>
</table>

The following table will serve as a template for the description of development activity within the overall context of the system’s development.
2.3 Spiral 1: Prior Development

Previous work on implementing image processing with a focus on hierarchically scaled aggregation began in 1993 when Dr. R. Ford completed the original version of “Merge”. This proof-of-concept prototype demonstrated that it was feasible to use the aggregation approach for the segmentation of very large images. The original system, implemented in the programming language Ada, could handle images up to a size of roughly 2,000 X 2,000 pixels. This version used a separate implementation of the “Classify” operation, written in FORTRAN by Z. Ma, which could scale to any size image. Continuing into 1994, J. Guo re-implemented Merge in C++, and was able to successfully process images up to 8,000 X 8,000 pixels. Z. Ma subsequently implemented a version in C that combined classification and aggregation, still focusing on processing 8,000 X 8,000 pixel images.

Both J. Guo and Z. Ma followed the basic approach taken in R. Ford’s original prototype, improving performance by carefully engineering use of auxiliary data structures. In 1995/96 S. Barsness created a C++ version of the aggregation process that took a novel approach to key processing aspects, producing a robust system capable of processing 12,000 X 12,000 pixel images. Barsness’s version, called “MegaMerge”, subsequently became the standard tool
for image segmentation used in land cover analysis in the national "GAP Analysis" project [Ford97]. S. Barsness also implemented an additional proof-of-concept system called "GigaMerge" that demonstrated that the merging of significantly larger images was also possible. GigaMerge, written in C++, successfully processed 30,000 X 30,000 pixel images and was reported to also be extensible to handle 100,000 X 100,000 pixel images.

In 1997, Professor Ford completed a research study that looked at the feasibility of porting the aggregation and classification operations to Java. The goal of this port was to obtain platform and system independence, because all existing versions had to be carefully tuned to the characteristics of different hosts. The initial study was successful: using mostly predefined object types and a relatively immature Java runtime environment, his initial Java version was still able to process images in the 4000 X 4000 pixel range.

Table 2: Spiral 1: Prior Development

<p>| Quadrant 1 | Title: Spiral 1: Prior Development (Gross Summary of Prior Merge Development Activity) |
| Description: Present the events leading up to personal involvement in Merge development activity. |
| Objectives: Implement H.S.A. for very large images. |
| Constraints: Known techniques do not scale to size needed. |
| Alternatives: Look at various programming languages, redefine the problem, and/or propose shortcuts to simplify the |</p>
<table>
<thead>
<tr>
<th>Quadrant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Risk Assessment: Solutions to implement H.S.A. could be programmed, but time and space of execution could prove prohibitive.</td>
</tr>
<tr>
<td>Risk Resolution: Build an object-oriented framework to implement a sequence of prototypes that seek to identify time and space bottlenecks. These components, interfaces, and object-oriented frameworks will then be refined to reduce risk/costs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quadrant 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions: Object oriented framework designed with components loosely defined as image objects with these characteristics:</td>
</tr>
<tr>
<td>• Image Input</td>
</tr>
<tr>
<td>• Area Identification</td>
</tr>
<tr>
<td>• Area Ordering</td>
</tr>
<tr>
<td>• Area Processing</td>
</tr>
<tr>
<td>• Image Output</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quadrant 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation: Met original goal of processing a large, 8000 x 8000 pixel image and the system became a standard used by USGS/GAP Project for land-cover analysis. Need a better user interface, portability needs to be enhanced, and greater modifiability to allow application to be run on various systems with multiple types of imagery. Also a better object-oriented structure should be implemented to allow components to be easily added/modified. Time and space performance needs to be enhanced.</td>
</tr>
<tr>
<td>Decision: A new “generation” of merge developers should be trained. Focus should be placed on a Java implementation to create a better user interface, increase portability, and create a better object-oriented design.</td>
</tr>
</tbody>
</table>
By mid 1997, multiple implementations of hierarchical scaleable aggregation were available and were being used in land cover analysis, petrographic image analysis, and other applications. However, there was no one implementation that could be easily adapted to both the needs of the user and the implementer when new applications were considered. Users were forced to adjust to the interface and parameters of existing programs, yet the programmers still had to resort to rewriting the major parts of the programs. It was obvious that a flexible graphical user interface would help accommodate the needs of users by offering an intuitive front end through which they could interact with the implementation. Further developing and refining the object structure of a suitable implementation could potentially allow for code reuse and modular code-hiding, allowing programmers to implement new changes with greater speed and accuracy. This is essentially the evaluation presented in Spiral 1: Prior Development, Quadrant 4 (see Table 2) and the point where the specific work of this project begins.

2.4 Spiral 2: First Implementation

During the fall of 1997 I was enrolled in CS541 Software Science I: Requirements and Specification, a class focused on the application of object-oriented software design principles. As a target for design
activities, we were introduced first to the aggregation problem from a theoretical perspective, then from the practical perspective of designing software to efficiently implement a merge system. While slightly confused as to the exact specifications of the target system, we were encouraged to start our first phase of the spiral. Prior development of this system was summarized in Spiral 1: Prior Development. The focus of the Fall semester was to train the students to be able to understand the information represented in Spiral 1: Prior Development, Quadrant 4 (see Table 2). This being a classroom project, many alternatives were mandated. In some cases even though the risks were judged by students to be unacceptably high, the decision to continue or terminate certain aspects of the project were often over-ridden by a higher authority (the professor).

While the objectives for this spiral were straight forward, "Implement a version of Merge in Java", there was much apprehension concerning the constraints listed in Spiral 2: First Implementation, Quadrant 1 (see Table 3). Due to this fact, much time was spent pondering and working out difficulties in Quadrant 2, such that all members involved felt secure enough to continue the project.
Table 3: Spiral 2: First Implementation

<table>
<thead>
<tr>
<th>Quadrant 1</th>
<th>Quadrant 2</th>
<th>Quadrant 3</th>
<th>Quadrant 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objectives:</strong> Implement a version of Merge in Java.</td>
<td><strong>Risk Assessment:</strong> Time not sufficient to develop entire Merge system in Java from scratch. Abilities and personalities vary amongst students, so intra-group conflict may prevent successful completion of project. Without full understanding of the Java programming language and the implementation details of merge, a new system cannot be completed within the given time limits.</td>
<td><strong>Actions:</strong> Incrementally, an image reader, area finder, area sorter and area processor were created. Team members met weekly to discuss progress and delegate tasks.</td>
<td><strong>Evaluation:</strong> Met original goal of implementing a version of Merge in Java. Implementation is inefficient and cannot process large images. An understanding of how the merge system works was reached, but the program is not</td>
</tr>
<tr>
<td><strong>Constraints:</strong> A working version has never been implemented in such a short time period. Problem is still not well understood by “new” implementors. Must work in groups.</td>
<td><strong>Risk Resolution:</strong> Implement several “toy” programs in Java to gain familiarity. Make incremental changes to existing code. Hold weekly group meetings to ensure group performance.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
completely functional as is.

Decision: Refine the system to be more efficient and reliable with user feedback and a graphical user interface.

Amazingly all teams were able to successfully create working implementations within the specified time limit. While Quadrant 4’s evaluation states that the implementation was often inefficient, never before had anyone implemented a working version of hierarchical scaleable aggregation in such a short time period. It is believed that the elegance of Java as an object-oriented programming language, the sub-division of the implementation task into a series of modules in Quadrant 3, the ease with which each module could then be implemented as a Java object, and the ability to work as a team were all responsible for this dramatic achievement.

2.5 Spiral 3: Canonical Ordering

The next phase of algorithm development, described in the next spiral, was to investigate different schemes for performing on-the-fly aggregation of “To-Be-Merged” (TBM) areas. The goal was to find an optimal scheme in terms of speed, while remaining as close as possible to a “canonical ordering”. Canonical ordering is an ordering method defined by sorting first by sizes, and then, when sizes are equal, ordering by smallest start row and smallest start column. The idea of
canonical ordering is to guarantee that although two different algorithms might operate differently, if they both aggregate elements in a manner consistent with a standard order, they should always produce the same output, given the same set of processing parameters. Were this ordering not enforced, very fast algorithms that processed areas in a more convenient order could be developed; however, such algorithms would not generate consistent outputs and thereby could not be used across multiple data sets.

In Quadrant 3, three algorithms are implemented and compared. One was the algorithm used in all the versions of "Merge", another was a very conservative canonical ordering based algorithm, and the third was a slight modification of the original. In order for either of the later algorithms to replace the original, first they would have to be proven to truly follow canonical ordering, and second, they would have to run faster than the original. Profiling done to these three algorithms in Quadrant 3, led to Quadrant 4’s conclusions that because there was no significant decrease in execution time provided by the new versions, there was no need to prove that the two new algorithms follow canonical ordering.
### Table 4: Spiral 3: Canonical Ordering

<table>
<thead>
<tr>
<th>Quadrant 1</th>
<th>Quadrant 2</th>
<th>Quadrant 3</th>
<th>Quadrant 4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Title:</strong> Spiral 3: Canonical Ordering</td>
<td><strong>Risk Assessment:</strong> May not be sufficient time to develop any “significant” enhancements to Merge. When two or more people work on a project there is always the potential for conflict.</td>
<td><strong>Actions:</strong> Three different area operations were implemented. Empirical tests demonstrated that a proof was not necessary. Profiling was also performed to identify bottlenecks within the system. Team members met weekly to discuss progress, alternatives, and delegate tasks.</td>
<td><strong>Evaluation:</strong> Canonical ordering limits performance, but small deviations from it do not offer a justifiable performance increase. Profiling can be used to as a tool to not only</td>
</tr>
<tr>
<td><strong>Description:</strong> Refinement of previous work and exploratory probes into new unresolved problems.</td>
<td><strong>Risk Resolution:</strong> Implement the options with the least risk (short implementation time). Choose to perform a theoretical study on area “stacking”. If time permits, as a backup, complete an empirical profile of different implementations. Hold weekly group meetings to ensure group performance.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Objectives:</strong> Enhance or study the Merge algorithm.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
identify performance bottlenecks, but also discover exactly where one implementation differs from another. Please refer to Appendix B for a detailed summary.

Decision: Construct a formal proof that this strategy is optimal, research the possibility of relaxing the canonical ordering criterion in order to further optimize speed, and run truly comprehensive tests with multiple experiments on very large images.

The end result of the class exercise was that although a new version following an optimal canonical order was not developed, much insight was achieved into the general design of such algorithms, and into aggregation in particular. Before an improved algorithm could be developed, it is first necessary to fully understand the implications of canonical ordering, especially best and worst case scenarios. In order to do this, one must become very familiar with the theory of area aggregation from the identification stage through the aggregation stage. In addition, to perform an actual test, there must previously exist a proven algorithm which can provide a performance benchmark with which to compare results from any new version. If no such benchmark system exists, then there is no basis to establishing correctness of a new system except by theoretical proof. This is one reason for the decisions that were made in Quadrant 4. Please refer to Appendix B for a detailed report of the activity performed in this spiral.
2.6 Spiral 4: Prototype GUI

An on-going concern over the large number of files generated by the Merge family of implementations had not yet been addressed in a previous spiral. In the next phase of development, focus was taken away from theoretical and performance issues and directed towards this increasingly bothersome problem. While the original objectives were as stated in Spiral 4: Prototype GUI, Quadrant 1 (see Table 5), those objectives were modified throughout the risk assessment and resolution stages of Quadrant 2, leading to the development of a graphical user interface to serve as the front end of a modified version of a general aggregation system. The primary reasons for developing a GUI were first, to automatically supply file names, thereby meeting the original objectives, and second, to make the implemented software more accessible to users with limited computer knowledge. The actual design ideas of the GUI varied between team members, so as part of the risk resolution in Quadrant 2, each member decided to implement his/her own ideas individually. The process was successful, as can be seen by the results in Quadrant 3.

Table 5: Spiral 4: Prototype GUI

<table>
<thead>
<tr>
<th>Title:</th>
<th>Spiral 4: Prototype GUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>First Prototype GUI</td>
</tr>
</tbody>
</table>
Quadrant 1
Objectives: Develop a method of handling the numerous filenames that are generated by each step of the Merge process.
Constraints: A short development time limit. Must work in groups.
Alternatives: No practical alternative.

Quadrant 2
Risk Assessment: May not be sufficient time to develop and implement a solution. When two or more people work on a project there is always the potential for conflict.
Risk Resolution: Start working fast. Each team member is responsible for his/her own work, and we will put on a semblance of group cooperation only during presentations.

Quadrant 3
Actions: A “universal code engine” was created so that a Graphical User Interface (GUI) and a Textual User Interface (TUI) could access the same code. To do this the code was revised to throw exceptions that both a GUI and TUI could catch. To demonstrate this a modified TUI and a completely new GUI were created. Refer to Appendix C for a description of the artifacts created.

Quadrant 4
Evaluation: While the new GUI proved to be useful and even helped with the file naming problem by automatically supplying names, it did not address the problem of organizing the numerous names that were generated.
Decision: Further work should be directed towards creating one unified program, not various programs combined under an interface, and the problem of numerous files still needs to be addressed.

When the project director was informed of the change in objectives, to create a GUI, he specified that the new implementation must also incorporate a traditional textual user interface (TUI). However, to have
a separate GUI and TUI would mean having a maintenance disaster anytime a change was necessary. The solution was to incorporate the use of exceptions everywhere a meaningful event occurred, so that both the GUI and the TUI could simply be event handlers attached to the same source code. The prototype that was created effectively handled the filenaming and GUI-TUI interaction problems, but as stated in the evaluation section of Quadrant 4, it did little to solve the organization aspect of having numerous files. Appendix C further discusses the details of this spiral.

2.6 Spiral 5: Beta Quality

As a direct result of the work on and subsequent evaluation of the prototype GUI it was next decided to create a production quality implementation of the aggregation system with a full-functioning GUI, and with elements that addressed the file organization problem. The constraints were again straightforward, as listed in Spiral 5: Beta Quality, Quadrant 1 (see Table 6), but the alternatives indicated another sub-objective that must be considered. Ideally, the new implementation would offer an entire array of basic image processing features required by a user who wants to analyze and manipulate digital imagery, such as cutting and pasting, regions of interest, drawing, pixel manipulation, and others.
### Table 6: Spiral 5: Beta Quality

| Quadrant 1 | Objectives: | Unify all merge algorithms into one coherent system. Resolve problems arising from numerous file names. Develop a stable GUI. |
| Quadrant 2 | Risk Assessment: | End users will not be satisfied with program. Program will not be competitive with commercial packages. The file naming issue may be very difficult to solve. The GUI must be easily extendible for future additions. |
|            | Risk Resolution: | Develop a series of prototypes to obtain user feedback. Identify public domain algorithms that can be used to give the system the basic functionality included in commercial graphics packages. Utilize tree hierarchies to organize files. |
| Quadrant 3 | Actions: | Multiple prototypes were created and actual beta testing to determine user satisfaction is in progress. |
| Quadrant 4: | Evaluation: |  |
|            | Decision: |  |
The system described in subsequent chapters meets these constraints. The risk assessment made in Quadrant 2 has been primarily resolved. Currently the development process is teetering between Quadrant 3 and Quadrant 4. As soon as feedback is obtained from system users, then this spiral will enter its evaluation stage. Since this spiral represents a great deal of time, and multiple sub-spirals are nested within it, the original risks may no longer be the current risks. Chapter 3 discusses some key risks that occurred, and further adds depth to this spiral.
3. Specific design and Implementation Considerations

As part of the actions performed in Spiral 5: Beta Quality, Quadrant 3, it was determined that construction of a series of prototypes would be the best way to identify the real risks and also elicit response as to the real wants and needs (objects). A combination of rapid prototyping and incremental model development was used as part of this inner-spiral. The incremental method is illustrated in the following diagram.

![Rapid Prototype Development Model](image)

**Figure 2: Rapid Prototype Development Model**

The basic GUI developed in Spiral 4: Prototype GUI, was used as the starting point, with the plan to refine additional versions from this
base. However, many problem areas existed in the GUI that had to first be resolved. First, I needed to understand more about the development of readily, easily extendible interfaces and usable environments. Second, I had to better understand how to incorporate basic image processing features into such environments. Third, I had to address the growing problem of how to manage and organize files to simplify the cumbersome process of user controlled file naming and organization. I knew that file organization must somehow tie-in with the object-object relationships inherent in image operations, but I still needed to determine the exact relationships and how they could be depicted in the system.

3.1 Property Files

At first the GUI was adapted so that it could be built automatically from a “properties” file. The properties file is a simple text file that lists what should appear in the menus and what procedures should be called when menu items were selected. Use of a properties file dramatically simplifies GUI-related (event) programming. For example, the addition of a function could be accomplished by adding one line of text in a properties file. Here is some sample text from the final version actually used:

```plaintext
# Plug-ins installed in the File/Acquire menu
acquire01 = "DICOM...",ij.plugin.DICOM
acquire02 = "BMP...",ij.plugin.BMP
acquire03 = "FITS...",ij.plugin.FITS
```
# Plug-ins installed in the Edit/Options menu
options01 = "Line Width...", ij.plugin.Options("width")
options02 = "JPEG Quality...", ij.plugin.Options("quality")
options03 = "Fonts...", ij.plugin.Fonts
options04 = "Debug Mode", ij.plugin.Options("debug")

Lines that start with a `#` are ignored as comments. To add an additional function to the Edit/Options pull-down menu, all that needs to be done is add the line `options05=` followed by the text to appear on the menu, and then the class and options to be invoked when that item is selected. Additional experimental versions also allowed the creation of new pull-down menus, renaming of existing menus, and other options, but my evaluation indicated that allowing the user to customize the interface options to such a degree added too much confusion by eliminating a standard look-and-feel for the application.

3.2 Basic Image Processing

After studying how users actually work, it became apparent that additional basic image processing functionality would need to be provided. A user may view an image, crop it, perform an aggregation process, and then want to zoom in on a particular portion of the resulting image. Having to implement these "standard" image manipulations is something necessary and time consuming, but neither interesting nor new. Including these operations also shifted the system concept from that of developing a standard version of the aggregation
process to that of developing an image processing environment that included aggregation. To avoid reinventing the wheel, I began to look for public domain image processing software with freely available source code. The search uncovered ImageJ, a Java implementation of NIH Image, which "is a public domain image processing and analysis program for the Macintosh.[NIH]" While still in its infancy, ImageJ had many of the basic image processing functions already developed, it is written in Java, and its source code is readily available.

ImageJ was studied from the inside out to understand its organization, communication, and functionality. The original ImageJ consisted of over 150 separate classes. The developer wanted to make it easy for users to add their own functionality to the program, so ImageJ supported so-called plugins, i.e., separately supplied classes that conform to a standard interface so that they can be easily incorporated in and called from an existing system. I created several sample plugins, but soon realized that the aggregation process would have to be embedded within the program to be fully functional. Part of this has to do with the problem of how to handle naming, as explained below.
3.3 File Naming Problem

One of the biggest problems with the aggregation process is that it uses and generates numerous files: at least an input image, similarity matrix, color palette, and output image. If input images were processed only once with fixed parameters the file naming would be manageable, but frequently a given input is processed several times with different parameters, producing several groups of outputs. Add in variety in pre and post aggregation processing, and it becomes very important to be able to determine exactly how a given derived image was produced. I first considered automatic naming conventions, along with a reference database, but eventually discarded these in favor of an object-origintree. In the same manner of a genealogy tree, every derived dataset can be placed as a node on the tree; the branches leading to that node indicate which functions with which parameters were used to produce that artifact.

3.4 Objects

Objects, the building blocks of object oriented programming, are entities encapsulating data and procedures. These entities allow an object to have its own internal state and behave autonomously from the program module in which it is created. This allows the users of the object to focus on what the object does, without needing to understand how the object does it. A class is a template for creating
objects. By reusing existing class specifications and extending them through class inheritance, new classes, and thus new types of objects, can be created. The graph of classes showing the interconnected inheritance relations is called a class hierarchy. This same model can be used to describe image–function relationships and deal with file referencing in the target image processing system. This method was implemented, using nodes interconnected in a tree-like structure, as described below.

3.5 Nodes

A node is different from an image file (image) because it contains additional information about the image. Then is a node simply the combination of an image and a file descriptor? No. While information can be saved in a file descriptor, the node, because it is an object, contains mechanisms for operating on itself. For instance, display mechanisms are present for image files, mechanisms for viewing, editing, and checking are present for similarity matrices, etc. All this functionality is “contained” within the node, because the node is an object for which this functionality is defined. A description of how nodes appear in actual use is included in Appendix D.1: Features of jMERGE (v0.826).
Nodes in the object-origintree are not loaded or saved in the traditional sense, because they are not separate files. Once a node is created it is part of the object environment, so a node cannot be easily deleted because it represents the history and current state of the environment. If deletion is allowed, then all descendants of the deleted node must also be deleted. Since confusion may arise over this, and learning from one's "mistakes" is important, explicit deletion is not allowed. However, the uncontrolled proliferation of nodes can also cause problems, so a mechanism to hide unwanted non-leaf nodes is included. This mechanism can also be set to auto-hide mode, where unwanted nodes will automatically disappear from the display (but not from the environment). However, the invisible nodes can be made to reappear if needed.

I realized that real-world users needed "open" and "save" work so that they could work in disjoint sessions and share work on separate machines with other people. The system was already designed so that there was no need to save when exiting the program, because the program always maintains its state and continues from its previously state. I added an import/export function to allow the entire state to be exported (saved as a serialized object) and then imported into another new environment in which it appears as a new root node. This approach copies the environment's information, but it is preferred
because the other alternative would be to attach it as a sub-tree off of the original root, or elsewhere in the tree. Having the system open to the new environment is no different than a split screen view of all existing environments, with the tree for one particular root node maximized in view. A problem exists with adding “sub-trees” in that their attachment to any node in an existing environment would indicate that that node, whether it be the Root or any other ancestor node, did not actually create the sub-tree. It would be possible to add an import node beneath the root node, and then attach the sub-tree, but that would still imply a stronger relationship then what actually exists. An additional problem would be that imported nodes would have to be checked for coherency problems if added to the original tree. Similar names, duplicate nodes representing the same images, and other such problems would have to be averted. The beauty of the program as a descriptive language is that is does not require a parser or lexical analyzer, which would be necessary should importation of sub-trees be allowed within an existing environment. As built, trees are always correct because they can only be built from nodes, and each node has state and only offers functionality that corresponds to its current state.

By duplicating the object-origintree, root and all, no relations are assumed between any non-linked nodes, which greatly reduces
complexity. Deletion of a node was finally permitted, but only in the most complete and destructive instance: the total deletion of the entire environment, reverting back to the original environment containing just a single root node.
4. Summary and Conclusions

The spiral model of software development explained earlier was used to organize the past two years worth of work that has gone into the development of the current implementations of the aggregation system. Of specific importance was the new Java implementation with the addition of a graphical user interface (GUI) to help unify previously separate algorithms into one coherent program. What originated, as a typical GUI, was refined to the point where the user interface became a mirror image to the underlying object structure. Similarly the implementations of the aggregation algorithms were refined into a comparable framework. The resulting “object oriented system” helped to alleviate problems in file naming and understanding of file and image relationships, as well as eliminate the need for traditional user interface features such as “undo” and “save”.

While this version met all of the original objectives of Spiral 5: Beta Quality, Quadrant 1 (see Table 7), concerns for beta-user familiarity and knowledgeable future programming support have led to increased efforts in the development of a traditional-style user interface over this experimental object oriented GUI. The traditional version, minus support for file relationship organization, has been completed and is currently undergoing actual field-testing.
Appendix D.2 contains an informal interview with a beta user. Most of his comments indicate approval with the general features presented in Appendix D.1. Potential areas for improvement lay in upgrading the tree drawing and refreshing algorithms, updating the code to a newer JDK, and possibly devising an additional menu system to help users migrate to the system. The objectree concept was well received and the use of serialized objects instead of image (tiff, gif, jpeg, ...) file formats was also appreciated.

Once the formal field trail reports are collected, Spiral: Beta Quality will enter its fourth and final quadrant, in which all the prototypes developed thus far are evaluated. The evaluation not only considers beta user input, but also programmer input, end-user demand, and various risk factors. Out of this will come suggestions for those who will continue with aggregation system support in the next spiral.
Appendix A.1: Definition and Description of Rule-Based Image Aggregation


A specific definition is necessary to ensure correct implementation and to compare result-equivalent, yet implementation-different methods.

We start by assuming that the primary input is a classified image in which each pixel has a single data value drawn from a discrete range of class identifier values \([0,...,C_{\text{max}}]\), where 0 represents a null-valued pixel that does not participate in the area aggregation process (e.g., a “border” pixel introduced in geo-rectification), and no other class values have particular numerical significance. To simplify the analysis, we also assume that both image pixels and the pixel grid are square, so that problem size \(N\) implies a square image with \(N^2\) pixels (and also a maximum number of \(N^2\) areas). We formalize aggregation as a function \(A(I_{\text{in}},\text{MMU},N,\text{O},T,X) \rightarrow I_{\text{out}}\), where

- \(I_{\text{in}}\) is the classified input image,
- \(\text{MMU}\) is the minimum mapping unit threshold,
- \(N\) is a key indicating which pixel neighbor definition to use,
- \(\text{O}\) is a TBM ordering function,
- \(T\) is a target neighbor selection function,
X is a set of exceptional survivor cases,

And I_{out} is the classified output image.

The resolution of areas in the output image is determined by the value of MMU and a set of special cases X. A survivor area in image II is one with size greater than or equal to MMU or which satisfies a case in X, whereas a to-be-merged area is one with size less than MMU which does not satisfy any case in X. For convenience we refer to the set of all survivor areas as SURV and the set of all to-be-merged areas as TBM. Operationally, aggregation begins with area identification, which relies on the exact definition of a pixel's neighbor given by N. Aggregation proceeds with a partitioning based on MMU and X that forms the sets SURV and TBM. Next, TBM areas are to be systematically eliminated, one by one, a process that involves the TBM ordering function O and the target neighbor selection function T: O provides a total ordering on the elements of TBM that defines TBM elimination order. In each elimination, T is used to select one of the neighbors of the TBM area as the target area which will absorb the eliminated area's pixels. The result of this of this process is the output image I_{out}. By distinguishing the neighbor definition, ordering function, target neighbor selection function, and exceptional processing cases (i.e., N, O, T, and X) as additional inputs we can distinguish secondary, application-dependent effects from the basic
properties of aggregation. The following properties summarize the intended behavior of the aggregation function.

Repeatability. The aggregation process will produce the same value if repeated multiple times on identical inputs.

Survivor Invariance. The class value of any pixel $P(I,j)$ in a survivor area in $I_{in}$ is unchanged in $I_{out}$.

TBM Elimination-1. There are no TBM areas in $I_{out}$.

TBM Elimination-2. The class value of any pixel $P(I,j)$ in a TBM area in $I_{in}$ may differ from its value in $I_{out}$.

Nested Hierarchical Scaling. Consider $A(I_{in},MMU1,N,T,X) = I_{out1}$ and $A(I_{in},MMU2,N,T,X)=I_{out2}$ for $MMU1<MMU2$ – some area boundaries of $I_{out1}$ may be deleted in $I_{out2}$, but $I_{out2}$ introduces no new area boundaries not present in $I_{out1}$. 

Appendix A.2: Advantages of Merge

By: Dr. Ray Ford, University of Montana

The process [Hierarchically Scaleable Aggregation] has two key characteristics that set it apart from other more traditional filtering and reduction-via-clustering approaches. First, the classification process, which is numerically intensive, is done only once, producing (in near-real-time) a base-line classified image with a minimum unit size of 1. Next, multiple derived datasets can be produced from this base map in parallel, aggregated at different minimum unit thresholds. Each dataset is guaranteed to have a hierarchical scaling property based on the unit threshold. It is easiest to describe this property in the special case of 2-D datasets where the units in question are areas. The hierarchical scaling property guarantees that for $X > Y$, when the areas in a derived image with unit threshold $X$ are compared with areas in a derived image with threshold $Y$:

- every "border" in $X$ is present in $Y$, and
- some borders in $Y$ are "erased" in $X$ (i.e., borders are eliminated and pixels in certain areas re-classified to aggregate small areas into larger areas)

Data reduction via a process that guarantees hierarchical scaling is particularly significant in terms of the faithfulness of data zooming. That is, you implement zooming by switching from an image with a larger unit threshold to a smaller unit threshold, both which can be
pre-computed, not by totally re-rendering the original dataset with
new depiction parameters. Using hierarchically scaled, aggregated
images introduces "virtual boundaries" that dissolve when the data is
depicted at higher resolution (lower minimum unit size thresholds),
rather than the "false boundaries" that move or change when other
more traditional re-scaling techniques are used. Essentially, the
hierarchical scaling property is a unit-based process, rather than a
pixel-based process, thus guaranteeing some integrity in how the units
(objects) in the original image are collected and depicted as coarser
(larger) units in a lower resolution depiction. Though the difference
may not be apparent in the initial depiction, it becomes apparent in
subsequent refinement via zooming.

The second significant factor in this approach is the
computational advantage it offers over other more traditional
approaches. It collects the most computationally intensive process,
that of initial classification (which requires some sort of statistical
clustering analysis) into a single process at the beginning of the data
processing pipeline. All subsequent data manipulation, via
aggregation, is handled in discrete processes that can be performed in
parallel for different minimum size thresholds. The aggregation
algorithm itself is be bound by memory constraints that are a function
of image size, not numerical processing constraints. Careful
engineering of the aggregation algorithm allows processing an $N \times N$ image in $O(N)$ space. In practical terms, this has allowed us to implement near-real-time processing of images for ecosystem modeling applications, ranging in size up to $100K \times 100K$ pixels with up to seven data values per pixel (i.e., with size ranging from 10-100 Gigabytes), using only simple single processor workstations with only 512Mbytes of process space.
Appendix B: Canonical Ordering Report

The following work was done in cooperation with Bill Zollinger, as part of CS542: Software Science II, Spring 1998/99 at the University of Montana.

Our project was to investigate different schemes for performing on-the-fly merge of TBM areas. The goal was to find an optimal scheme in terms of speed, while remaining as close as possible to the canonical ordering, defined as ordering by size first, then, when sizes are equal, ordering by start row and start column. We investigated three methods: two discussed in class, and a third that we devised ourselves. Initially, we had planned to construct a formal proof that we had an optimal algorithm; this was abandoned in favor of a more empirical testing strategy for several reasons. First, it took us some time to identify the need for and then develop a "clock-free" timing mechanism for Merge. Secondly, due to the fact that jMerge is an evolving project, we needed time to stabilize a current version with which to work. We had to analyze the code extensively to determine whether it actually performed as we expected it to. Lastly, we did not have a successful implementation of an algorithm using the canonical order, implying that we did not have an output image merged using the canonical order with which to compare our output.
We were told that it was desirable to remain faithful to the canonical order, but that speed of execution was really the most important factor. If, by diverging from the canonical order, the speed could be greatly improved, then this would be a desirable direction in which to concentrate future effort. Therefore, we had to first get a speed estimate for our algorithm before we could even ascertain whether a proof of optimality was necessary. Only if it were significantly faster than the existing implementation would this be the case.

Strategy number 1 was, after finding areas beginning in row $K$, to merge areas of size $N$ beginning in row $K-N$. This was the original Merge strategy and it was proved in class not to be consistent with the canonical ordering. Strategy number 2 was to merge areas of size $N$ beginning in row $K - \left( \sum i \right)$, where $i = 1$ to $N$. This was devised to cover the extreme case of having an area of size 1, with an area of size 2 stacked on it, with an area of size 3 stacked on that, etc. This was a “conservative” scheme, and while an official proof has yet to be made, it is conjectured to be consistent with the canonical ordering.

Strategy number 3 was our strategy; it is a slight modification of strategy number 1. In this scheme, areas of size $N$ are merged beginning in row $K - (N+1)$. This is based on the theorem that an area can be merged ahead of another provided the two areas are not
neighbors. An area of size $N$ beginning in row $K-(N+1)$ will have at
least one space between it and any area beginning in row $K$, and so
this condition is the one that guarantees that the areas in question are
not neighbors, while maintaining minimum separation between them.
We believe this strategy to be consistent with canonical ordering,
though we did not undertake a formal proof.

Our performance results were mixed. First, there is an
implementation error in Strategy 2 in which not all TBM areas were
merged. We discovered this problem arose when searching for
"mergeable” areas. Secondly, we discovered that the total time to
execute is essentially a meaningless measure of execution speed
because it includes the user-interaction segment, where the
parameters for Merge are entered by hand. Therefore, unless
otherwise noted, all time will be area identification/merging times only.

The test image we used was 888 x 838 pixels, with an MMU of 40.
All three algorithms took roughly the same amount of time
(approximately 4 minutes, 7 minutes with profiling). The output
images were converted to ASCII format with IUTIL and found to be
different, which means that Strategy 1 uses to implement on-the-fly
merge certainly does matter. Strategies 1 and 3 produced the same
number of survivor areas, but the number of total areas was differed
by one.
A profile was run in an attempt to determine the methods most often called within the program. From this it is even easier to see that something is wrong with Strategy 2. The most frequent call, java/util/BitSet.get(), is called approximately 40000 times less than in the other two strategies. Comparing the two methods which actually merged all the TBM areas, Strategy 3 made approximately 0.004% more calls to BitSet.get() than Strategy 1. Implementation of Strategy 3 is recommended because it is believed to be consistent with the canonical ordering, whereas Strategy 1 is known to not be consistent. In addition, MERGE_MGR.MERGE() itself is called approximately 350 times less in Strategy 2, which corresponds to the difference in the number of surviving areas found by the other two strategies.

We have found a strategy that we believe to be both consistent with the canonical ordering required by the specification, and only slightly slower than the current implementation (which is not consistent with the canonical ordering). We believe, but have not proved, that this strategy is actually optimal amongst consistent strategies. Possibilities for further study are: actually constructing a formal proof that our strategy is optimal or near optimal, research into the possibility of relaxing the canonical ordering criterion in order to further optimize speed, and running truly comprehensive tests with multiple experiments on very large images.
Appendix C: Prototype GUI Report

Original Goals:
One of the original goals for this project was to develop a method of handling the numerous filenames that are generated by each step of the "Merge" process. A method for organizing the file structure was to be proposed and implemented. In addition, a prototype graphical user interface (GUI) was to be implemented for demonstration and evaluation. This user interface would include any changes proposed to handle the file naming problem.

Modified Goals:
A GUI would be developed for the JUTIL and AREAOPNS portions of the merge project and an analysis of the over-all system structure would be made to see if there was a way to have a textual user interface (TUI) access the same functional code that the GUI would access.

Progress:
The overall system structure of the merge project was studied and a method to create a "universal code engine" devised, so that both the TUI and GUI can access the same functions. This method has been implemented and there is a working prototype of the GUI and TUI accessing the same code. In order to accomplish this in a useful and
easily expandable manner, much revision had to be done to the
system structure of the merge project. Also, in order to demonstrate
that a TUI and GUI could access the same code base, a GUI had to be
created. All of this has been accomplished and details are contained
within the following sections of Appendix C.
Appendix C.1: System Structure of Original Merge and First Prototype

This section contains diagrams of the original merge system structure and the first major prototype's system structure. Also included are sample screen dumps from the first GUI created. No sample output is presented for the TUI because its output is almost identical to the original output from the JUTIL program. The pages are organized in the following format:

- Original Package Structure (1 page)
- Original Use Diagrams (3 pages)
- New Package Structure (1 page)
- New GUI Package Use Diagram (1 page)
- New OPTION Package Use Diagram (1 page)
- New EXCEPTION Package Hierarchy (4 pages)
- New GUI Sample Screen Shots (7 pages)
Original jMERGE
Package Structure

```
d.newest.jmerge
```

DEFAULT PACKAGE
Original jMERGE
Use Diagram 1

- AREAOPNS
- BITSTUFF
- ERDAS
- TIFF
- IMAGE
- POINT
- PTQUEUE
- PALETTE
- OBSAVER
- SIMPLE_IN
- SEGMENT
- BITMAP
- EXECTIME
Original jMERGE
Use Diagram 2
Original jMERGE
Use Diagram 3
Classify and Misc Relations

Diagram:
- EXECTIME
- CLASSIFY
- SIMILARITY
- SIMPLE_IN
- MERGE_MGR
- AREA
- AREA_QUEUE
- PTQUEUE
- ERDAS
- TIFF
- IMAGE
- PALETTE
New jMERGE
Package Structure

GUI

TUI

EXCEPTIONS

OPTIONS

FUNCTIONS
New jMERGE GUI Package Use Diagram

- mainGUI
- ASCII_TO_ER_GUI
- ERDAS_HDR_GUI
- ER_TO_ASCII_GUI
- TIFF_TO_ASCII_GUI
- SEGMENT_GUI
- mergeGUI
- classGUI

- ASCII_TO_ER
- ERDAS_HDR
- ER_TO_ASCII
- TIFF_TO_ASCII
- TIFF_TO_ER
- SEGMENT

jMERGE.OPTIONS Package
New jMERGE
OPTION Package
Use Diagram
File Exception Heirarchy

To increase font size the following abbreviations were used:
FNF = File Not Found
FF = File Format
FC = File Creation
Rcb Exception Heirarchy:
Row Exceptions

RcbException

Row Exception

RowInvalidException

StartRowInvalidSmallException

EndRowInvalidException

RowTooSmallException

StartRowTooSmallException

EndRowTooSmallException

RowTooLargeException

StartRowTooLargeException

EndRowTooLargeException

Col Exception

Band Exception

JMergeException
Rcb Exception Heirarchy:
Column Exceptions

- JMergeException
- RcbException
  - Row Exception
  - Col Exception
    - ColInvalidException
      - EndColInvalidSmallException
    - StartColInvalidException
      - StartColLessThanEndColException
  - Band Exception
    - ColTooSmallException
      - StartColTooSmallException
    - EndColTooSmallException
    - ColTooLargeException
      - StartColTooLargeException
    - EndColTooLargeException
Rcb Exception Heirarchy:
Band Exceptions

- JMergeException
- RcbException
  - Row Exception
  - Col Exception
  - Band Exception
    - BandInvalid Exception
      - StartBandInvalid SmallException
      - EndBandInvalid Exception
    - BandTooSmall Exception
      - StartBandToo SmallException
      - EndBandToo SmallException
    - BandTooLarge Exception
      - StartBandToo LargeException
      - EndBandToo LargeException
Default Screen

Optional Screen Representation with Enabling Methods Shown
Run Options

File Options
Utilities Options

Window Options
Sample Open File Dialog
Sample dump of log file to text area after displaying ERDAS header
Classify Window
Merge Window
Appendix C.2: Details of First Prototype Package Structure

To distinguish this version from earlier versions, the system was renamed jMERGE and broken up into multiple packages in order to emphasize the need for modularity, encapsulation, and code hiding. A GUI or TUI has a certain look and feel, but that look and feel should not be tied directly with the actual image processing being done behind the scenes. The GUI or TUI should be able to be modified decoratively without having to worry if the system’s functionality will be compromised. In addition, it is necessary that the GUI and TUI process information in the exact same fashion, and that if a change is made to a low level processing function, both of the user interfaces would have immediate access to the new changes. The original merge structure could not support any of these needs, while the new structure supports all of these needs by adding a layer of abstraction between the user interfaces and the functional code.

All user interface related code is contained in either the jMerge.GUI or jMerge.TUI packages. These packages both have access to the jMERGE.OPTIONS package. JMMERGE.OPTIONS contains generic services (not dependent upon a particular user interface), each of which provide a desired functionality. However, the classes in jMERGE.OPTIONS do not contain the low level processing services.
Those services are provided by classes in jMERGE.FUNCTIONS, because low level services may be frequently changed or updated through optimization. If such a change were made, only classes in jMERGE.OPTIONS would need to be modified, if any at all. Since jMERGE.GUI and jMERGE.TUI only access jMERGE.OPTIONS and not jMERGE.FUNCTIONS, the GUI and TUI will both accommodate the new changes instantly without any modification.
Appendix C.3: Details of First Prototype Exceptions

The jMERGE.EXCEPTIONS package, and numerous exception classes were created in order to increase the flexibility and error handling of all classes, especially those in the GUI. A GUI has to be able to deal with errors (exceptions) in a graceful and meaningful manner. For example, if the user enters an illegal filename, and the filename was one of five that was entered, then rather than having the program terminate, it would be desirable to have a message box pop up asking the user to re-enter just the one incorrect filename. By having a large hierarchy of exceptions it is possible for one part of the system to throw an exception indicating exactly what file was not readable and why, and for the GUI to catch that exception and deal with it appropriately. If, on the other hand, it is appropriate for the system to halt with any error (as in portions of the TUI), then the specific exception does not have to be caught, and instead only the presence of any exception needs to be noticed. In other words, when an exception is generated, the most specific exceptions is thrown, but only the most general exception that a handler wants to deal with needs to be caught.
Appendix C.4: Details of First Prototype Option Package
Standard Class Interface

Each of the classes in jMERGE.OPTIONS has a standard base interface. It consists of a null constructor, a constructor that accepts a Vector, a GO() service that returns a Vector or throws exceptions, and a GetOutput() service that returns a Vector. Here is a sample:

```java
public ASCII_TO_ER(){}
public ASCII_TO_ER(Vector files)
public Vector GO() throws ErdasFileFormatException,
                        ErdasFileCreationException, AsciiFileNotFound,
                        AsciiFileFormatException
public Vector GetOutput()
```

The Vector passed to the constructor is a Vector of parameters that are required to perform useful services. The reasons the parameters are passed as a Vector is first, it makes for a tidy interface, and second, and more importantly, so that if the class is ever changed, there will not be a parameter count or type mismatch. In addition, the class would be able to be modified to accept and process requests from old clients based on its “previous” structure, while providing new functionality to only those clients who send a Vector of the correct larger size.
The service GO() is called when the client is ready for actions to be performed. The reason the constructor and GO() were separated is that a Vector needed to be returned, and a constructor can not return a value. The Vector returned by GO() is a Vector of strings representing a log of what actions the service completed. It is left to the discretion of the client to use the log or not, but it can be very helpful when an exception is thrown and the user or programmer wants to know how much of the service was completed before the exception occurred. If GO() throws an exception then the running log will not be returned. That is why the GetOutput() service must be present. If the client catches an exception, it can request the log from GetOutput() and it will contain information pertaining to the events surrounding the generation of the exception. This information can be used in addition to information supplied by the exception, or it can simply be dumped for the user to view.
Appendix C.5: Details of First Prototype GUI

The GUI was created using the Java AWT 1.1 classes and the ActionListener event model. From the default or optional screens, the user can click on either the menu selections of the quick launch buttons to activate a service. Most file and utility options are based upon the sample file dialog. Input file or files are requested, then an output file is requested, and finally the request is processed. Any errors that occur are displayed in a message box as well as dumped to the text area of the default screen.
Appendix D.1: Features of jMERGE (v0.826)

Environment: Graphical Image Processing Environment

jMERGE is a graphical image processing environment that visually and automatically manages all images, files, and relationships between all images and files. Every artifact (an image or file) is represented as a circular node that is drawn on the screen. These nodes are connected by lines (links) that signify the relationship between those nodes. These nodes and links are drawn in a tree structure with the oldest/original node appearing at the top of the window and the newer nodes appearing at the bottom.

- Straight Line Links

  If two nodes are connected by a straight line then the visually "lower" node is a child of the visually "higher" parent node. This means that the parent node under-went some transformation, the result of which is represented by the child node. This type of line represents a parent-child relationship.

- Double Line Links

  If a double line connects two nodes then the visually "higher" node was used in the creation of the visually "lower" node, but is not the parent. This type of line represents a user-used relationship.
• Circle Nodes

A normal node that represents some artifact and contains information as to how it was created as well as other information.

• Square Nodes

Similar to a circle node except that one or more of its "children" are currently hidden (not visible on the screen). If the user clicks on a square node a pop-up menu would list the option to Un-Hide Children. By selecting that option, a square node will turn into a circle node and all of its children will be displayed on the screen.

• Auto-Hide

This feature goes along with square nodes, in that all unimportant nodes (level of importance can be set by user) that are not leaf-nodes (a leaf node has no children) are automatically hidden from view to reduce clutter. The parent of the automatically hidden node is transformed into a square node.

• Background Saving

When a node is created a multi-threaded process automatically saves the new artifact. This is done transparently (in the background) so that the user can continue working without
having to wait for the artifact to finish saving. If at any time the user wants to restore the artifact or possibly compare a new image with an old image he/she simply clicks on the node and selects the option "view image". Then the image as it was at the time the node was created will be loaded and displayed in a new window.

- **Undo**

  Undo operations are not required because every action was recorded and all artifacts were automatically saved. If the user wants to back up and start over again, he/she simply clicks on the node that represents the state he/she wants to start over from.

- **Typical Graphical User Interface (GUI)**

  Most features that users would expect to find in a modern GUI are available. Menus, toolbars, quick-access keys, pop-up menus with currently relevant options, progress bars to indicate length of delays, message boxes, and other features are all available. All menu commands are multi-threaded, meaning the user can run several different processes at the same time if so desired.
Appendix D.2: Interview with a BetaUser

This is a transcription of an informal interview that took place with beta user (n001) on November 23, 1999.

I: Interviewer
B: Beta User

I: What are your general impressions of the system you’ve been testing?
B: Which one? jImage or jMerge.
I: What is the difference?
B: jMerge is the one without a menu bar and instead has a node tree, whereas jImage is straightforward Image/J with aggregation plugins and filters.
I: Tell me about your impressions with jMerge.
B: It’s amazing it works, but the interface is definitely cool. It’s got its fair share of rough edges, but, hey, it’s a prototype, what do you expect?
I: Tell me something about the rough edges.
B: For one, every time the tree gets updated, the screen will flicker. Also, if you hide several nodes on one layer, and then have a parent node produce some more offspring, then the tree is no longer centered in the window.
I: Any guesses as to why that might be occurring?
B: The screen flicker is definitely the whole area getting refreshed, instead of just updating modified nodes. As to the centering problem, it’s actually centered if you included the positions for the hidden nodes.
I: Any other problems?
B: Well, it's not really the fault of the program, as much as the fault of the JDK that was used, but the cursor will sometimes freeze in popups [menu] that contain scrollbars. I'm more or less into new interfaces so for a proto [prototype] I didn't really notice too many major problems.

I: So, do you mean that the software has problems, but that you do not consider them to be problems?

B: Only having popup [menu] appear from nodes, and no pulldowns [menus], might be a problem for someone who only knows [MS] Office, but if you're open to new ideas, I think that it's actually cool, because you have all of your available options right where you're using them. Probably helped the programmers too.

I: You seem to like the interface. What other things do you like about the software?

B: The nodes and images are always updated. If you click on an image, that image's node gets selected as well. Also, since it saves everything as serialized objects, it actually makes reading and writing faster. It's got some options to customize the environment, so that once you get used to things, you don't have to let the repetitive graphics distract you. You know what? Macros would be nice, because then you wouldn't have to click on the same options over and over again.

I: So it has no macro facility?

B: No, there's a separate textual input method, that you can script, just no visual macros. I guess it wouldn't really matter with the images I'm doing, because it takes such a
long time between operations, that I’d rather run a script and just walk away. The naming is pretty cool too.

I: The naming of what?

B: The basic system functions, like instead of Export or Copy, its called Clone. You just clone your program. Or say you want to delete everything you’ve done so far. Then choose the Armageddon option, and blast it to pieces.

I: Sounds fun. How do you feel about the lack of open/save operations?

B: I don’t care. I’ve been working by myself, so I just turn it on, and everything is just as I left it.

I: What about limited deletion?

B: I don’t know about you, but I don’t make mistakes. No, really, I was once told that a mistake is just another learning opportunity. If you’ve learned from your mistakes and don’t want them around, or if the boss is coming, then just hide them. Same difference.

I: That’s one way to look at it. Could you make me a few screen shots?

B: No problem, ... [small talk edited out]
Bibliography


