Gradual Awareness Notification for the Desktop Environment

by

Thomas Blake Wilson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 26, 2006

Author .....
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by .....
Robert C. Miller
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Gradual Awareness Notification for the Desktop Environment

by

Thomas Blake Wilson

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis develops and puts forth the principles of *gradual awareness notification*. It distinguishes the concepts of *hard* and *soft* notification, and defines situations where gradual awareness techniques can be of the most benefit. Furthermore, it applies gradual awareness principles to the desktop environment to produce *slow-growth notification*, a visual notification system relying on slowly growing windows. It describes the design principles behind gradual awareness notification, and presents a prototype implementation of slow-growth notification called the *Slow-Growth Library*, or SGL. Finally, it presents user study results which indicate that slow-growth notification can achieve significant benefits over traditional popup notification systems without needing to be informed of the user's current task. The study demonstrates that slow-growth notifications were up to 39% less disruptive than popups, and up to 33% subjectively less annoying.

Thesis Supervisor: Robert C. Miller
Title: Associate Professor

Acknowledgments

First, I would like to thank Professor Rob Miller for his patience and insightful advice. The many conversations over the past year and a half were integral to establishing, presenting, and polishing the ideas found in this thesis, in addition to being exciting and intellectually stimulating. Without his assistance and guidance, this thesis would never have come into being.

I would also like to thank the members of the User Interface Design group at MIT. Their support, both in the form of being test users and in their valuable feedback and advice, played a crucial role in the process of developing this thesis.

Additionally, Radhika Jagannathan has been a source of constant comfort and friendship during the often trying journey of writing this thesis. Her patience in listening to my frustrations and rantings was unparalleled, and greatly appreciated.

Finally, I would like to thank my parents for their constant support and encouragement throughout my time at MIT, both as an undergraduate and this past year. Without them, I never would have made it through my undergraduate career, let alone finished this thesis. Thank you so much for everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As users grow more and more dependent on computing systems, the demands on a user's attention grow dramatically. In fact, user attention may be rapidly becoming the scarcest resource in a computing environment [7]. Many applications seek to notify a user in the course of their daily tasks. Email clients, instant messaging systems, background system tasks, firewalls, and virus scanners are just some of the programs that might need to inform a user. Figures 1-1, 1-2, and 1-3 illustrate a sample of the range of notifications that may appear, both centrally and peripherally. When the number of applications clamoring for attention is small, the task of managing all of their interruptions is not particularly onerous. However, when large numbers of applications demand attention within a short time span, the user can rapidly become overwhelmed.

In attempting to understand and improve notification, this thesis defines the terms *hard* and *soft notification.* Current desktop notifications tend to be hard, meaning that they interrupt the user suddenly and without regard to the user's current task. This leads to many problems. First of all, much research has shown that interrupting users at the wrong moment results in a severe loss of productivity. [6] Because hard notification schemes are usually unaware of the user's task, they often interrupt at sub-optimal moments. Some work is being done on notification schemes that attempt

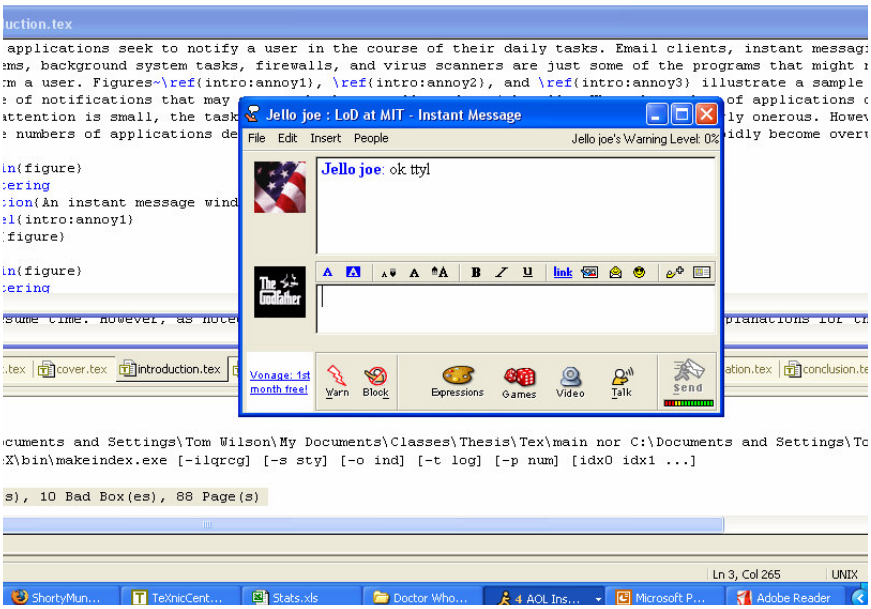

Figure 1-1: An instant message window interrupting a primary task

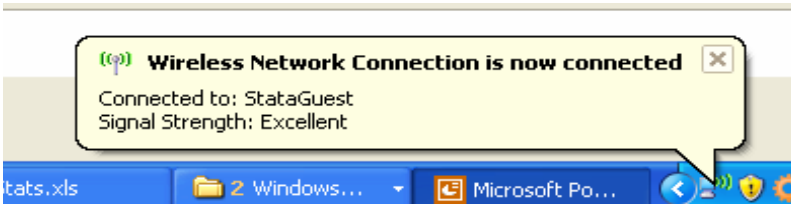

Figure 1-2: A network status notification

Figure 1-3: An incoming email notification

to make informed choices about interruption timing [18, 19], but these systems are complex and far from finished. Second, hard notifications tend to increase levels of user annoyance [5]. The lack of control and proper timing can cause a great deal of frustration to the user.

Soft notification, in contrast, is a method of notifying users through the use of gradual changes. Rather than using sudden changes, soft notification gradually changes information in the periphery of the user's awareness. Thus, soft notification may also be referred to as *gradual awareness* notification. In general, the principle of soft or gradual awareness notification is to start a signal at an imperceptible level and gradually increase the intensity of the signal until the user notices. This may involve any or all of the user's channels for input (visual, audio, tactile, etc).

As an example, consider a soft notification cell phone. Instead of ringing or vibrating at full intensity, a "slow phone" could start by vibrating very softly. Gradually, the intensity of the vibration would increase, up to some maximum level. At some point, the audio channel could also be employed. The phone would begin ringing softly, slowly increasing the volume of the ring tone. This alerts the user to the phone's attentional demand gradually, without necessarily breaking their concentration. (For a similar and entertaining line of research, see Marti and Schmandt's squirrel phone [21])

Of course, sometimes it is necessary or even desirable for the phone to forcibly interrupt a user (eg, to remind the user that they're about to miss their plane, or sleep through a meeting). In these cases, soft notification is clearly inappropriate.

This thesis presents an application of soft notification to the desktop environment. In particular, it presents a system of notifications that can be incorporated into existing or future applications. Since the primary information channel on the desktop is visual, the technique presented in this thesis focuses on a visual method. This method is hereby referred to as *slow-growth notification*, or just *slow-growth.* Slow-growth works using windows that start out infinitesimally small. Gradually, these

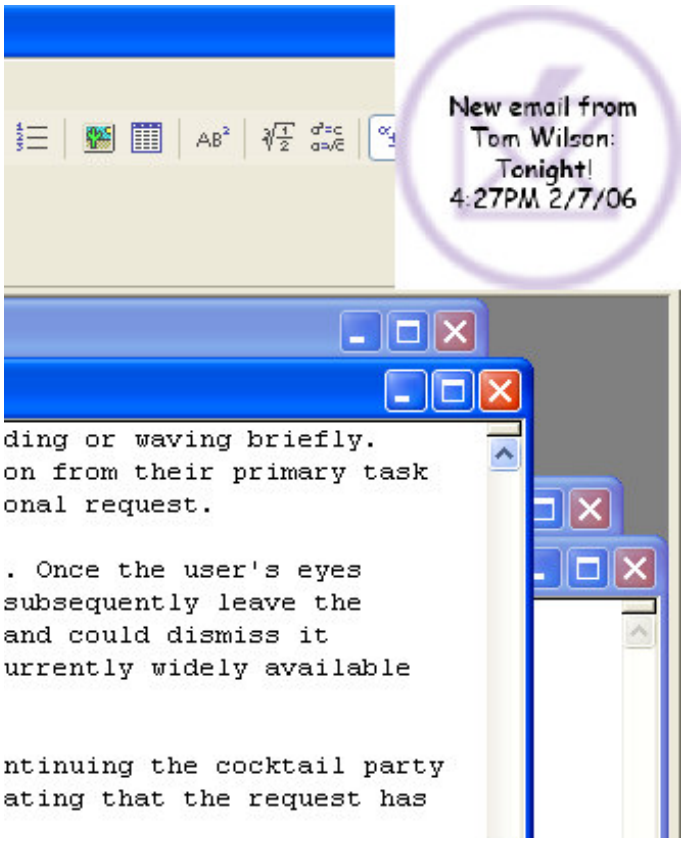

Figure 1-4: A slow-growth window in action

windows grow into the user's field of vision.

Slow-growth is interesting because it effectively *circumvents* the problem of creating an informed notification system. Rather than attempting to build a cognitive model of the user's task, slow-growth notification exploits the user's own internal model. As discussed in Chapter 2, a user's attentional field enlarges at natural task breaks. This leads to users noticing the slow-growth notification at *a moment at which they are more ready to be interrupted.* Thus, all the benefits of an informed notification scheme can be achieved without the implementation complexity or training phase associated with informed notification. Note that the goal of slow-growth is not to mediate notifications for existing applications, but rather to provide a tool to allow designers to more effectively and considerately notify users.

1.1 Contributions

This thesis seeks to prove the following: **Gradual awareness notifications will be consumed by users at natural task breaks, improving performance and reducing annoyance when compared to traditional hard notification schemes.** In the course of demonstrating the above claim, my thesis makes the following contributions:

- It develops the general principles of soft notification, and presents a workable conceptual model for gradual awareness notification systems.
- It describes and demonstrates a practical prototype implementation for such systems, based on slowly growing notification windows which gradually encroach on a user's awareness.
- It presents the *Slow-Growth Library* (SGL), a toolkit for using slow-growth notifications in applications.
- It presents quantitative data showing the performance benefits of slow-growth notification as compared to popup-based notifications.

1.2 Overview

Chapter 2 presents an overview of previous work in related fields. Chapter 3 describes the taxonomy of notifications developed for this thesis, and identifies key categories for which gradual awareness techniques are ideally suited. It also examines situations where gradual awareness may not be appropriate. In Chapter 4, I describe the user interface and methods of interaction for the Slow-Growth prototype. This chapter also identifies the crucial challenges in designing a gradual awareness notification scheme, and proposes a solution. Chapter 5 examines in-depth the actual implementation of the Slow-Growth prototype system, and describes the design trade-offs that were

considered during the development process. Chapter 6 presents the findings from the user tests I conducted. The results of both a field study and a lab study are described here. Finally, Chapter 7 restates the contributions of this thesis, draws implications from the results presented in the rest of the document, and proposes directions for future work in this area.

Chapter 2

Related Work

This section presents an overview of research in the fields of interruptability and vision/cognition. *Interruptability* is generally the problem of determining when and how to present interruptions. The problems of interruptability most relevant to this work include understanding the costs of interruption, mitigating those costs, isolating reasonable interruption moments, and deciding how to effectively display information to the user. *Vision* and *cognition* are concerned with understanding how events are processed by the human optical system, and how these stimuli are propagated to upper levels of the mind. In particular, vision and cognition research illustrates an interesting difference between what we *see* and what we *notice*.

Much of the previous work in this space has focused on techniques for mediating interruptions from many applications via generalized interruption managers. This thesis focuses on a slightly different approach by giving designers tools to improve the effectiveness of notifications in their own applications. However, it draws on techniques developed in previous research, and thus it is instructive to examine prior work on this topic.

2.1 Interruptability

2.1.1 Costs of Interruption

An interruption can be defined as *an event which requires a user to refocus their attention away from their primary task.* Interruptions are often annoyances which can adversely affect a user's experience. However, sometimes the information contained in an interruption is important enough to the user that the annoyance can be justified. In attempting to enhance a user's experience, it is useful to have a notion of what the cost of interrupting a user might be.

The costs of interruption are well-studied. Horvitz and Apacible present a useful study of previous work, along with techniques for measuring and predicting interruption costs [16]. They use predictive models built up over time to estimate expected interruption costs. Bailey, Konstan, and Carlis directly measure some of these costs [4, 5]. Their studies show that interruptions have a negative effect on both task performance and a user's emotional state. Task performance was significantly slowed when tasks were interrupted. Additionally, users reported feeling significantly higher levels of annoyance and anxiety when their primary tasks where interrupted. Research also indicates that these effects are dependant upon the user's mental load at the time of interruption [5].

Bailey and Konstan present a study of the impact interruption can have on user performance and satisfaction [7]. Their results indicate that when "peripheral tasks interrupt the execution of primary tasks, users require from 3% to 27% more time to complete the tasks, commit twice the number of errors across tasks, experience from 31% to 106% more annoyance, and experience twice the increase in anxiety than when those same peripheral tasks are presented at the boundary between primary tasks."

Additionally, the natures of the primary task and the interruption both influence the disruptiveness of the interruption. For example, Maglio showed that continuously scrolling displays can be considered more disruptive than discrete displays when the

primary task is a continuous word editing task [20]. Czerwinski, Cutrell, and Horvitz investigated the particular interruption of instant messaging (IM) windows [12]. They demonstrate that sudden popup notifications (like the type used in most IM systems) are particularly damaging and disruptive to fast, stimulus-driven tasks.

Finally, other work suggests that the effectiveness and disruptiveness of interruptions depends highly on the modality of the interruption. Most information in the computer desktop space is conveyed by visual means, with occasional auditory cues. This includes information from both primary and peripheral tasks. However, as demonstrated by Bodnar, Corbett, and Nekrasovski, there are certainly other modalities that can be exploited [10]. They were able to show that the olfactory channel is less effective at notifying users, but produces significantly less disruption. It is interesting to consider the possible notifications that might be designed using other modalities, such as olfactory or tactile.

2.1.2 Mitigation

Given the potentially high costs of interruption, it is desirable to understand how to mitigate the impact of an interruption. There has been an increasing amount of research in this area, and several basic techniques have been developed.

One simple technique for mitigating interruption costs is called *bounded deferral*. Bounded deferral is the policy of delaying notifications for some maximum time bound if the user is busy when the notification arrives. The idea is that for most tasks, users will become free with high probability in some time bound that can be learned [2]. The bounded deferral manager that Achlioptas and Horvitz describe uses a prediction function to estimate the "business" of the user, and delays notifications until the user becomes less busy (according to the predictor), or some maximum time bound is reached.

The bounded deferral technique is interesting, but its effectiveness depends on the validity of the predictor function. Constructing good predictors may require

knowledge of the user's task, or it may require a training phase to collect information and learn patterns. These factors necessarily make the system described in [2] a system which places an additional burden on the user, making it less appealing for widespread adoption.

Another interesting system making use of inference models and predictors is *attention-sensitive alerting* [17]. An interesting notion put forth in this work is the idea of balancing the interruption cost with the content of the notification. In particular, they recognize that a user may be more willing to pay a higher interruption cost to receive a critical notification than they would for a trivial one. Again, though, they make use of inferred models which require training.

The most recent technique for mitigating the costs of interruption is based on the work of Bailey and Iqbal [6]. Their results indicate that the cost of an interruption is intimately tied to the mental workload of the user. Their work concludes that interruption costs can be mitigated effectively by interrupting users at moments of low mental workload. This paper also presents their finding that mental workload is minimized precisely at "subtask boundaries" [6]. Similar work is presented by Adamczyk and Bailey, which demonstrates further that interruptions at subtask boundaries are significantly less disruptive [3]. This is the principle that motivates this thesis. Slow-growth notification attempts to exploit moments of low mental workload to deliver notifications while minimizing disruptiveness.

2.1.3 Isolation

The problem with the mental workload strategy is that isolating low workload moments is, in and of itself, a difficult task. In general, the problem of *isolating* ideal interruption moments is a challenging one. Several systems have been proposed to solve this challenge.

BusyBody [18] attempts to determine ideal moments for interruption by building up task models from training data. The system requires a training phase, during

which users are periodically asked to assess their levels of interruptability. This data is combined with a logged stream of desktop events to produce Bayesian nets for estimating ideal interruption times. Once again, though, this system is heavyweight in that it requires a training phase, which places a significant burden on the user.

Another system called MeWS-IT [19] exploits the mental workload principle described in the previous subsection. The approach in MeWS-IT is two-fold. First, it attempts to build task models for common tasks offline, using a system called the Task Model Builder. This uses representative user interactions to attempt to model common tasks. These models are then fed in to an Interruption Manager, which attempts to understand where the user is in the model at any given moment. Interruptions are then delayed until boundaries in the precomputed task model. This approach clearly has great promise, but its effectiveness is intricately tied to the validity of the precomputed task models. This may require building up an unmanageably large set of task models in order to be generally applicable.

Yet another class of system is in development. These systems attempt to use physical sensors to determine the user's availability. Several implementations of such systems have been described [9, 14]. These systems have certain advantages over pure desktop solutions. They are able to capture a more complete picture of user availability, as opposed to interruptability. Sensor-based systems can capture physical, real-world events, such as conversations, phone calls, or the user's physical absence. These indicators can used to more accurately determine when to interrupt the user. Unfortunately, these systems place a significant additional burden on the user, as they require additional hardware sensors. This makes them impractical for wide scale deployment.

2.1.4 Notification

Finally, one last problem in the field of interruptability research is the issue of *notification,* or how to actually convey the desired information to the user in an efficient

manner while minimizing disruption.

McCrickard and Chewar put forward the principle of attention-utility tradeoff: "The success of a notification system hinges on accurately supporting attention allocation between tasks, while simultaneously enabling utility through access to additional information" [22]. The concept is a useful one to consider when attempting to design notifications. In order to preserve the user's attentional focus, it is desirable for notifications to be minimally disruptive. However, in order to actually provide utility to the user, the notification must contain sufficient information to allow the user to act on it appropriately.

Extending this work, McCrickard, Chewar, Somervell, and Ndiwalana present the IRC framework for evaluating notification systems [23]. The three criteria they present are *interruption*, *reaction*, and *comprehension*. These three factors provide a logical, descriptive way to categorize and evaluate notification systems. Interruption is defined as an event requiring the transition of attentional focus away from the primary task, and has been covered extensively in this section. Reaction is taken to mean the accurate and timely response to the stimuli of the notification. Finally, comprehension is defined as the user's ability to remember the information for later use. This framework allows meaningful discussion of the various goals of notification systems. In particular, slow-growth notifications trade reaction speed for minimizing interruption. For further discussion of this topic, see Chapter 3.

Another particular area of notification systems are peripheral display systems. These systems attempt to provide information in a format that be easily absorbed in a glance. The fundamental principle of peripheral displays is that they sit in the user's peripheral vision, and can be checked quickly and easily. Examples of such displays include things like stock tickers, headline banners, clocks, battery indicators, etc. Other systems are more complicated peripheral systems that strive to keep the user informed of more complicated events [13, 26]. Maglio and Campbell analyze the impact of different designs on the effectiveness of peripheral displays [20]. Their

results indicate that it is possible, by careful design, to reduce the disruptiveness of a display without reducing its effectiveness.

2.2 Vision and Cognition

Vision and cognition research has interesting impacts for the problem of user notification. These two fields study what we *see* and what we *notice*, respectively. Although these two concepts might seem identical, they are in fact very different.

Chun and Wolfe describe how the brain responds to visual stimuli [11]. "At any given time, the environment presents far more perceptual information than can be effectively processed. Visual attention allows people to select the information that is most relevant to ongoing behavior." Effectively, the brain is doing filtering to determine which pieces of the myriad incoming visual stimuli we actually process and can respond to. Chun and Wolfe also describe the effect of cognitive "tunnel vision." Under heavy mental workload, a user's effective field of vision is decreased to the point where all they can perceive is a small area of focus directly related to their task. Once their mental workload decreases, their field of effective vision broadens again.

This interesting bit of neuroscience leads to some interesting phenomena such as *change blindness*, where users simply don't notice gradual changes in the environment. Simons and Chabris demonstrate change blindness convincingly [25]. Change blindness occurs because of visual attention filtering and cognitive tunnel vision. In Simon and Chabris' study, users were asked to watch a video and keep count of the number of times a ball is passed. This task required a high degree of focus and placed users under heavy cognitive load. Users were so focused on their task that most failed to notice changes in the environment, including an actor dressed as a gorilla passing through the frame.

Gradual awareness notification seeks to utilize this feature of human visual process-

ing. By only making gradual changes in the environment, gradual awareness notifications avoid impinging on the central areas of the user's task. Thus, the notifications are not noticed until the user's mental workload decreases, allowing their field of effective vision to expand.

Chapter 3

Taxonomy of Notification

Notifications are used for a wide variety of purposes in modern technology. Different systems inform the user in different ways, according to the context in which they operate and the information they wish to communicate. When considering how to notify a user, the designer must take these factors into account. Different types of notifications are appropriate in different situations. Before attempting to analyze the benefits of gradual awareness notifications, it is useful to consider which circumstances are most appropriate for this form of notification. Figure 3-1 presents a taxonomy tree for different classes of notification. This section attempts to define these major classes of notification, and evaluate the applicability of gradual awareness techniques to these domains.

3.1 Synchronous

Synchronous notifications occur in direct response to a user action. As an example, consider the default behavior of the Firefox browser. When a user attempts to close a Firefox window containing multiple tabs, the browser displays a warning box asking if the user wants to continue. Figure 3-2 shows this behavior. Another example is a common and useful feature in word processors. If the user tries to close the window

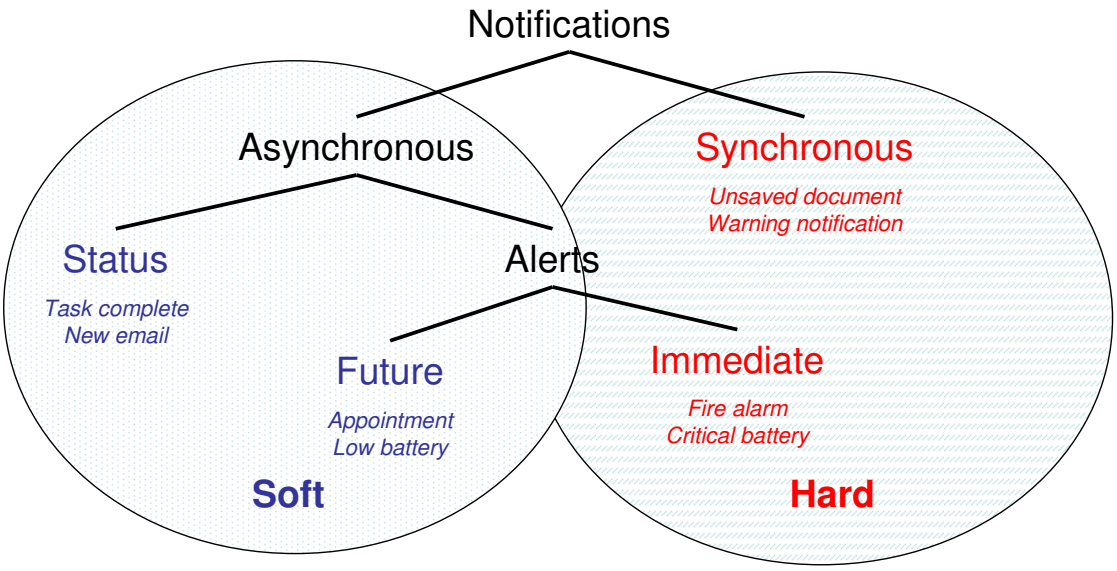

Figure 3-1: Notification taxonomy

without saving the current document, most word processors will display a warning asking if the user wishes to save before closing. Figure 3-3 illustrates this behavior.

The user has taken an action, and as a direct and immediate result, the application wants to convey information back to the user. In this case, a gradual awareness notification technique is not only inappropriate, it may be actively harmful. Use of a popup window or other such "hard" notification is desirable. In order to maintain perceptual fusion of the action and the notification, the delay between the two events should not exceed 100ms. With gradual awareness techniques, this is not likely to occur. Note that the timing of the notification is critical here. Because the notification occurs very shortly after the action, the user is unlikely to have moved on to another task, and thus hard notification avoids the traditional costs of uninformed interruption.

As an additional subtlety, any notification generated by the user's primary task should be treated as a synchronous notification. As long as the notification is coming from the user's primary task, minimizing response time becomes more important than minimizing disruptiveness. After all, it is not disruptive if the notification is

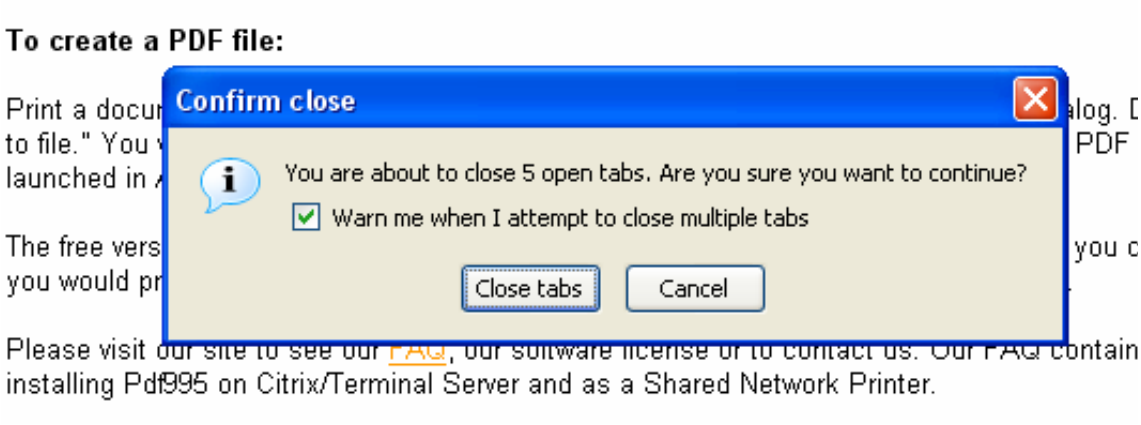

Figure 3-2: Firefox's default multiple tabs warning notification

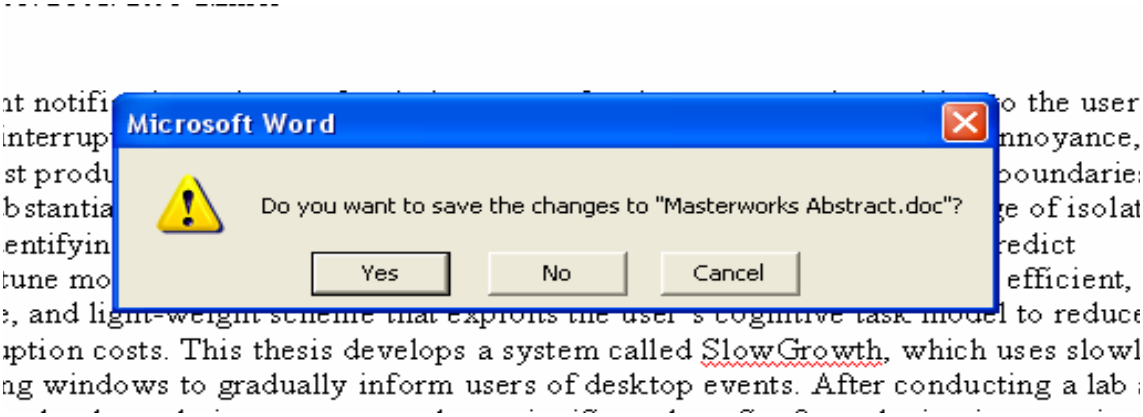

Figure 3-3: Word's unsaved document warning notification

part of the primary task. For example, if a user is attempting to load a webpage and the page cannot be found, it would be desirable to use a hard notification to alert the user quickly. If, however, the user has moved on to perform another task, a soft notification would be more appropriate so as not to interrupt the user's new task. Notice that this applies to the *user's* primary task, which is not necessarily the *system's* foreground task.

3.2 Asynchronous

Asynchronous notifications do not occur in direct response to user action. Instead, they are triggered by external causes. These causes may be predictable (such as a scheduled appointment) or not (such as an incoming email). In this case, the notification comes as a result of user action performed at a prior time, or as a result of some change in the state of the system. In either case, the notification is perceived to be a separate event from the cause of the notification. This is the domain in which gradual awareness notification can be of most value.

Asynchronous notifications can be further divided into *status* and *alert* domains.

3.2.1 Status

Status notifications convey information about changing state. In general, this category contains all notifications that reflect changes in information that the user cares about. Examples of this type of notification include new email announcements, "task complete" notifications for long-running tasks, or buddy list changes in programs such as AIM. These notifications occur at times which may be difficult or impossible to predict, which means that users are frequently in middle of another task when they occur. Figure 3-4 shows an example of a task completion notification interrupting the user's current task.

This is a domain where gradual awareness techniques can be of great value. Be-

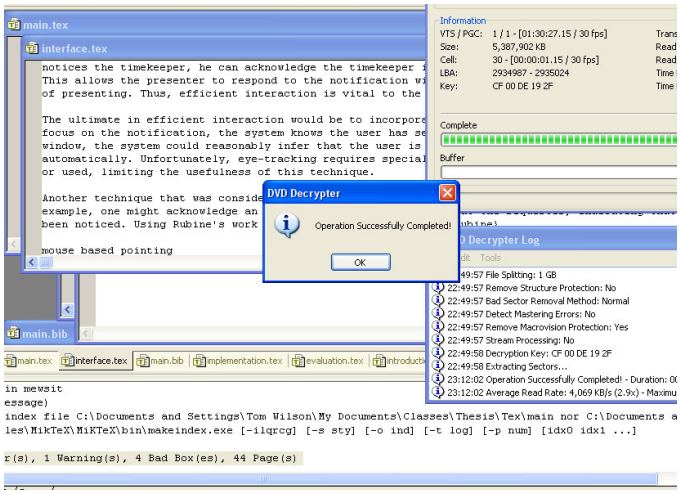

Figure 3-4: A typically intrusive task completion notification

cause the user may be in the middle of another task, interrupting them at the wrong moment may be very costly, as described in Chapter 2. Using a gradual awareness notification in this case causes the user to notice the notification at a more optimal moment with high probability, avoiding the cost of an ill-timed interruption without requiring any knowledge of what the user's task is. Because the user was not expecting the notification to come at any particular time, the cost of noticing it slightly later is likely to be small, as argued by Achlioptas and Horvitz [2]. This class of notifications is a perfect example of when it may be desirable to slow reaction time in order to reduce the cost of the interruption.

3.2.2 Alerts

Notifications which have time-sensitive information are categorized as alerts. Failure to consume the alert within a given time bound may be costly. Examples of alert notifications include battery warnings, calendar appointments, and clock alarms. In these applications, the time at which the notification is delivered is a vital part of the effectiveness of the application. For example, imagine a computerized fire alarm. No matter what else the user is doing, the fire alarm requires immediate action on the

user's part. It may not be appropriate for the designer of a fire alarm system to use gradual awareness notifications.

Alerts can be further subdivided into *future* and *immediate* alerts.

Future

Future alerts have deadlines that can be predicted in advance. For example, in a scheduling application, a user may wish to display a reminder when a meeting approaches. It does the user no good if they don't notice the reminder until after the meeting has started. But the application knows exactly when that is, and thus can predict when the user must notice the notification. Good examples of future notifications include scheduling notifications, clock alarms and battery life warnings.

Because the user may be doing some important task when the notification needs to be delivered, it is desirable to attempt to minimize the interruption cost of the future notification. However, the notification must be presented before the deadline so that the user can take whatever action is required by the notification. In order to balance these concerns, these notifications can take advantage of gradual awareness notification. By varying parameters of the notification itself (such as the initial delivery time and the rate of change), the designer can ensure that the notification reaches its maximum visibility at the deadline for the event. In SGL, this means varying the growth rate of the window such that the notification window reaches its maximum size at the deadline. Either the user will hit a natural task break before the deadline and notice the notification, or the deadline will arrive, causing the notification to reach maximum visibility, which should hopefully be sufficient to bring it to the user's attention.

Immediate

Immediate alerts are a different type of alert. These alerts occur at less predictable moments, and require immediate action. Examples include fire alarms, critical battery

warnings[1], uninterrupted power supply (UPS) warnings, and many more. Immediate alerts require action by a deadline which is not known ahead of time. This lack of knowledge means that immediate alerts occur very close in time to their deadlines. Thus, the speed of the user's reaction is essential to the effectiveness of the notification. Since gradual awareness techniques minimize interruption cost by reducing the speed of a user's reaction, these techniques are most likely not appropriate for use in immediate notifications.

3.3 Context Sensitivity

Although this chapter has presented a classification scheme for notifications, it is important to realize that the appropriate classification for any given notification is intimately dependent on its context. Many different factors influence what techniques are most appropriate to use for any given notification. For example, as mentioned in Section 3.1, whether a particular notification is coming from the user's primary task or from a peripheral task influences whether the notification should be hard or soft.

Classification becomes particularly difficult when considering the alert category: notifications may graduate from being future alerts to becoming immediate alerts as the situation changes. For example, when the battery is low but not critical, a battery notification may be appropriately classified as a future alert. However, when the battery reaches a critical level, it may be more appropriate to consider it an immediate alert. This is further complicated by the fact that factors beyond the application's scope may influence when the notification needs to promote from future to immediate. In the battery example, if a large number of actions need to be taken before the system can be safely shut down, the application needs to promote the

[1]Notice that battery warnings can be considered either future or immediate alerts, depending on the nature of the event. A warning such as "5 minutes of battery power remaining" is most likely a future alert, as the action time is some point in the future. However, a notification like "Critical battery level. Switch to AC power immediately" is more accurately considered an immediate alert. See Section 3.3 for more detailed analysis.

alert much earlier than the time of the battery's death. Another example would be in a scheduling application. A meeting reminder may become an immediate alert 5 minutes before it starts if the location is close by. However, if the user needs to drive across town, the notification needs to become an immediate alert much earlier.

In addition to context, the content of the notification has a direct effect on how to classify it. For example, consider a telephone call. If the content of the phone call is extremely urgent, the phone may wish to interrupt the user's primary task. If, however, the content is less important than the primary task, a softer notification would be appropriate.

As these complications indicate, automatically classifying notifications is extremely difficult, and far beyond the scope of this thesis. The techniques developed in this thesis are not attempting to classify notifications, but rather to provide a tool for developers to use, and to help them understand when to employ it. With proper task analysis and user studies, developers should be able to determine when it is appropriate for them to use hard notifications, and when the user would be better served by soft notifications.

Chapter 4

User Interface

For this thesis, the principle of gradual awareness notification was applied to the desktop computing environment, specifically the visual channel, to create a slow-growth notification toolkit called the Slow-Growth Library (SGL). This section discusses the user interface design for the SGL toolkit. It analyzes the important features of the user interface, and discusses trade-offs and previous design iterations that led to the current design.

4.1 Design Motivation

The original impetus for developing the technique of slow-growth notification arose from a desire to produce a less intrusive notification technique. As an example, consider a cocktail party. In order to gain someone's attention, a person could shout their name across the room, or grab their arm. However, this is disrespectful to the people the recipient is conversing with. It forces the recipient to abruptly interrupt their primary conversation to pay attention to the interloper. A more respectful action would be to request their attention by waving or making eye contact. Once the recipient acknowledges the request, they can finish or suspend their conversation gracefully, and smoothly pick up the new task with ease.

This contrast between a demand and a request is the fundamental design principle for the user interface in SGL. Current notification techniques tend to be hard, which makes them very intrusive for the user. These hard notifications constitute an attentional *demand*: pay attention to me now. Just as in a real life conversation, demanding a user's attention is highly insensitive to the user, and can create a great deal of annoyance.

A more appropriate paradigm for notification seems to be that of an attentional *request*: pay attention to me when you're ready to. Rather than demanding that the user shift focus from their primary task to deal with a notification, slow-growth notification attempts to request focus using gradual awareness. As explained in Chapter 2, the user will notice the request at a natural task break, allowing them to respond appropriately.

4.2 Features

In order for an attentional request to be functional, there are three characteristics it must possess. First, it must be *subtle*. Second, it must be *informative*. And finally, it must be *efficient* to interact with. These three principles form the basis for the user interface design used in the SGL toolkit. The following subsections define each principle in more detail, and discuss how they are manifested in the user interface design for SGL.

4.2.1 Subtlety

A demand can afford to be loud or obnoxious in order to attract the user's attention as quickly as possible. However, that very act of forcing the user to transfer their attentional focus creates a distracting situation and incurs all of the costs mentioned in Chapter 2. Therefore, it is important that a request remain *subtle* in order to minimize the impact on the user. In this context, subtle is taken to mean quiet or

gradual change. In a notification system, subtlety can be achieved by using gradual awareness: start by making a soft demand on the attentional channel (e.g., changing few pixels or using a low volume sound) and gradually increase the intensity.

A popup notification creates an attentional demand by changing many pixels at once. This sudden flash of change swiftly attracts the user's focus, interrupting them from the primary task. Similarly, a visual popup may be accompanied by a hard audio notification such as a chime or beep. These hard notifications force the user to drop their current task and attend to the source of the demand.

SGL achieves subtlety by using slowly growing windows. When an application triggers a slow-growth notification, a 1 pixel square window appears in the corner of the screen. The corner is used for several reasons. First of all, the corner is at the periphery of the user's view, and thus notifications here are less likely to interrupt the user's primary task. Secondly, the Fitts' Law target size for mouse-based interaction presented by a corner notification is effectively infinite in extent, which makes interaction more efficient. Finally, a particular peculiarity of the prototype implementation constrains the SGL prototype to the corners of the screen, as discussed in Section 5.2.1. This window gradually begins to grow towards the center of the screen at a developer-specified growth rate. Figure 4-1 shows a sample of an SGL notification as it grows. The number of pixels changing at any one time is much lower than in the case of a popup notification. Thus, the slow-growth notification is less likely to attract the user's attention if they are currently in the middle of a task. When the user reaches a task boundary, however, their field of attention widens, and they can notice the attentional request.

Clearly, the window's rate of growth plays an important role in how subtle the notification is. A popup window effectively has an infinite growth rate, which makes it highly distracting. However, a window which has a zero growth rate is unlikely to be noticed, and thus unlikely to serve the primary function of a notification. There is a tradeoff here, with some balance to be struck between response time and disrup-

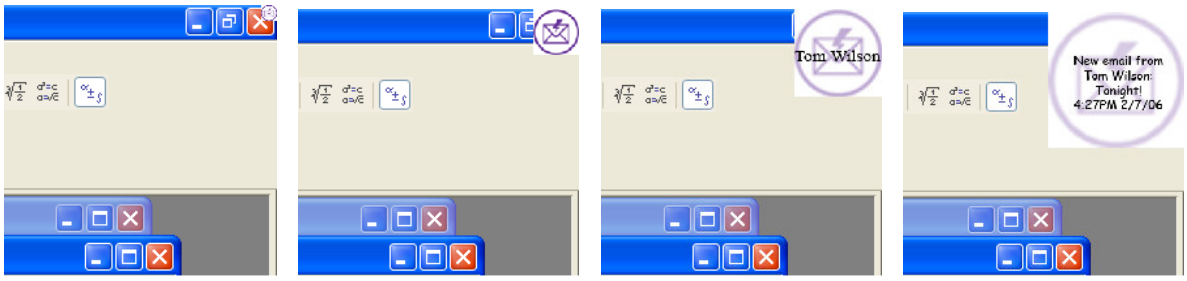

Figure 4-1: A slowly growing window

tiveness. SGL allows developers to set a growth rate which is appropriate for their application's needs.

Different growth rates may be appropriate for different contexts. Notifications which need to be noticed earlier may want to use higher growth rates, while notifications which can afford to be noticed later can get away with using slower growth rates. Using the taxonomy from Chapter 3, alerts with imminent deadlines may wish to use higher growth rates, while status notifications can use lower. In the experiments performed for this thesis, three different growth rates were used: 10, 5, and 1 pixels per second. For more details, see Chapter 6.

4.2.2 Informativeness

An attentional request should also be *informative*. The goal of an attentional request is to convey as much information as needed to the user in as efficient a manner as possible. Ideally, the user can learn all the information they need simply by glancing at the notification. Popup notifications have an advantage in this respect: because the size of the notification is known ahead of time, a designer can carefully construct the notification to ensure maximum informativeness. However, with slow-growth notification, an interesting challenge arises. Because the notification's display area continually changes in size, the amount of space a designer has to convey their message is continuously changing. The challenge is to design the notification such that it conveys as much information as possible in a usable fashion regardless of the

Figure 4-2: Continuous visual zooming

notification's size when the user notices it. In designing SGL, three different display modalities were created to solve this problem.

The first and simplest approach is called *continuous visual zooming*. In this technique, the designer simply creates the notification as it will appear when it reaches full size. The notification is then scaled to match the current size of the slow-growth window. Figure 4-2 illustrates this behavior.

Continuous visual zooming is appealing because it is simple. It is conceptually clear to understand, and it requires no additional work on the part of the interface designer. Continuous visual zooming is most useful when the notification design only contains elements that are clear even at small sizes (e.g., icons). The problem occurs when elements of the notification are hard to make out at small sizes. For example, continuous visual zooming is not particularly useful for notifications containing large areas of text. At small sizes, the text becomes unreadable, and thus useless to the user.

In order to solve this problem, the next approach implemented was called *discontinuous semantic zooming*. The concept of *semantic zooming* is to actually render the display differently at various sizes, and display different information at each size. [8] A good example of semantic zooming is the behavior of applications such as Google Maps. [15] At the high level, only major roads are shown. As the user zooms in on the location, minor roads and other geographical features suddenly appear in the display. The technique is referred to as discontinuous because the transitions between states occur suddenly and discretely. In the Google Maps application, the zoom bar is a

Figure 4-3: Discontinuous semantic zooming

series of discrete steps, rather than a continuous slider. This concept can be applied to slow-growth notifications, as shown in Figure 4-3.

The designer can design a notification for each of several different size levels (e.g., small, medium and large). For example, consider applying the discontinuous semantic zooming technique to a notification for an incoming email. The smallest size might simply display an icon indicating that the notification relates to the email client. Once the notification reaches a larger size, the designer might choose to display text indicating who sent the email. As the notification continues to grow, eventually it will reach a size where the designer is free to display more detailed information about the notification, such as the subject line and the time stamp.

Discontinuous semantic zooming may enhance the clarity of the notification at smaller sizes over continuous visual zooming. Because each level of the notification is designed for a specific size, the user is better able to gain information from the notification at small sizes. However, discontinuous semantic zooming is not perfect. First of all, it requires designers to create multiple versions of each notification they wish to use in their application (one for each size level desired). It is also less subtle, and may cause additional distraction for the user. Since changes in the body of the notification occur at discrete intervals, this may cause more pixels to change suddenly than would occur in continuous visual zooming. This "flashing" may prove disruptive to the user, interrupting them in the middle of a task.

The final technique developed for SGL is the most complicated. *Continuous semantic zooming* is a combination of the previous two techniques. It takes the basic

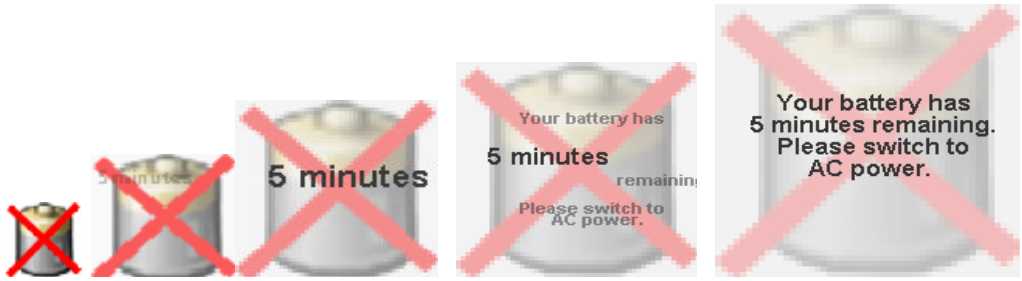

Figure 4-4: Continuous semantic zooming

approach of discontinuous semantic zooming, but rather than changing the content of the notification immediately at discrete steps, it uses a key-framing system to continuously animate between states. The designer specifies what the notification should look like at specific sizes along its continuum of growth designated as key frames. The system then smoothly interpolates between key frames to display the notification. Figure 4-4 demonstrates this technique in action.

Continuous semantic zooming allows designers to specify what the notification should look like at specific sizes, thus maintaining the clarity of discontinuous semantic zooming. It also uses interpolation to smoothly animate between states, eliminating the sudden jerks from the discontinuous technique. This allows it to maintain the same smoothness as the continuous visual zooming technique. Thus, continuous semantic zooming combines the benefits of both of the previous techniques to produce an highly informative and non-disruptive display. The trade-off, however, is an increase in complexity for the designer. In order to effectively use continuous semantic zooming, the designer must put additional effort into designing the intermediate states of the notifications (i.e., the key frames) as well as designing the final notification.

Recognizing that each technique may be appropriate for different purposes, SGL includes support for all three of these information display modalities. If the notification is simple and easy to understand even at small sizes, the designer may wish to use the simplest technique of continuous visual zooming. If the notification requires

different designs for different sizes, but the changes between size levels are small, the designer can use discontinuous semantic zooming without significantly increasing the distraction of the notification. Finally, if the designer wishes to create a smooth and non-flashing display, they can use the more advanced technique of continuous semantic zooming. In this way, SGL supports creating notifications that are as useful as possible to the user regardless of size in addition to providing flexibility for developers with differing needs.

4.2.3 Efficiency

The final characteristic of an attentional request is that it should be *efficient* to interact with. If the process of interacting with the notification takes a great deal of effort or focus, the user's ability to resume their primary task afterwards will be impaired. As an example, consider the action of a timekeeper at a presentation. When the presenter runs low on time, the timekeeper might wave at the presenter. Once the presenter notices the timekeeper, he can acknowledge the timekeeper in an efficient manner by nodding or waving briefly. This allows the presenter to respond to the notification without diverting much attention from their primary task of presenting. Thus, efficient interaction is vital to the success of SGL as an attentional request.

The ultimate in efficient interaction would be to incorporate some form of eye-tracking. Once the user's eyes focus on the notification, the system knows the user has seen it. When the user's eyes subsequently leave the window, the system could reasonably infer that the user is done with the notification, and could dismiss it automatically. Unfortunately, eye-tracking requires specialized hardware which is not currently widely available or used, limiting the usefulness of this technique.

Another technique that was considered for SGL is that of *gesture based input*. Continuing the cocktail party example, one might acknowledge an attentional request by waving at the requester, indicating that the request has been noticed. Using Rubine's work on gesture recognition [24], a system for recognizing mouse gestures

was constructed and added into the slow-growth prototype.

Unfortunately, there are some problems with adapting Rubine's algorithms to this problem. Firstly, the algorithms developed for gesture recognition assume a known start and ending point. In this domain, however, the start and end points of the gesture are unknown, since all the notification can capture is a continuous log of mouse movements. The biggest difficulty with gesture based input, however, is the problem of designing a gesture. In order to avoid accidentally triggering the gesture recognition with the user's normal mouse movements, a good gesture must be unambiguous. However, in order to be useful, a good gesture should also be simple and memorable. Designing gestures that meet both of these criteria is extremely challenging, and would have required too much time to implement. Note that this may be an interesting extension to the system for future versions.

The final interaction scheme developed for use in SGL is a form of point-and-click interface. When the user notices a slow-growth window, they may move their mouse over the window. This causes the notification to immediately expand to its maximum size. Thus, if the user notices the notification while it is still small, they are not forced to wait for it to grow to learn whatever information the notification provides. The user can then choose to dismiss the notification by clicking on it with the mouse, or they may choose to simply move the mouse out of the notification. This causes the notification to resume its previous size and growth.

At first glance, it may seem strange to ask the user to click on the notification to dismiss it, rather than simply interpreting moving the mouse out of the bounds of the window as a dismissal. However, because the notifications start out extremely small, it is quite possible that the user may place their mouse over the notification without realizing it was there. In this case, the user most likely wants to be able to continue their task without dealing with the notification. Thus, since the intention of moving the mouse out of the bounds of the notification can be ambiguous, the application makes a safe choice by doing nothing. This reduces the risk of unintentionally

dismissing windows, which may be costly for the user.

This interaction system allows the user to quickly and easily learn what information is conveyed in the notification, and to dismiss it simply once they have consumed the information. Notice that slow-growth windows can be dismissed by clicking anywhere within the extent of the window, as opposed to forcing the user to hunt for a specific target. This makes it much simpler and quicker for the user to dismiss notifications they no longer need. Of course, developers can freely place clickable targets within the notification in order to support additional functionality.

Chapter 5

Implementation

This chapter describes the details of the implementation of the SGL toolkit. It also discusses how developers can incorporate slow-growth notifications into their applications, and describes the particular mechanics of the system.

5.1 Overview

SGL is implemented entirely in Java, and is packaged as a JAR file. This allows other developers to add slow-growth notification to their applications simply and cleanly by importing the JAR file. Note that SGL provides a toolkit for developers to build less disruptive applications, as opposed to mediating interruptions from external sources. Because of this flexibility, it is simple for developers to use slow-growth notifications in conjunction with other notification techniques.

This section describes some of the key implementation details of the SGL toolkit. It explains how the system works, and examines interesting details of the architecture. The SGL system is divided into two major components: the SlowGrowth container, and a custom component called the NotifyImage. SlowGrowth is the wrapper around the control functionality of the system, while NotifyImage handles the details of displaying the notification's content.

5.2 SlowGrowth

The SlowGrowth component is the control wrapper for the system: it functions as a container for the NotifyImage, and handles all the spatial details of the notification's presentation, such as size and position. It also handles all of the user interaction with the notification framework (ie, the mouse-over interaction and dismissal). Finally, this component provides the entry point for developers to interact with the SGL system.

The SlowGrowth class is implemented as an extension of a undecorated JDialog. Thus, the standard desktop title bar does not appear on SGL notifications. This frees up screen space for the actual content of the notification. Also, since the standard title bar has its own minimum size, its absence means that slow-growth windows can start out much smaller. Additionally, this design allows SGL windows to appear without creating an icon on the task bar, which helps to minimize the total amount of change on the screen that slow-growth windows cause. The sudden appearance of a task bar icon would, in effect, create a hard notification, disrupting users. Finally, this design for the SlowGrowth component means that SGL windows can appear without stealing focus from currently active applications. This is a crucial point in the usability of SGL. If slow-growth notifications stole the application focus when they appeared, this would seriously damage the user's ability to continue work on their primary task while the window grows. Stealing focus would create annoyance, frustration, and confusion, especially if the user could not see the slow-growth window.

To configure the display of the notification, a developer must access the NotifyImage component contained within the SlowGrowth instance. See Section 5.3 for more details on the NotifyImage component. The SlowGrowth container affects the control parameters of the notification. These include the growth rate (how fast the window grows), the starting location (where the initial window appears), and the initial and maximum sizes for the notification. In order to adjust these parameters, the SGL API supports standard accessors and mutators for each parameter, allowing developers to

modify notifications as appropriate for their applications.

Once a SlowGrowth instance has been configured, the developer simply invokes the `start` method of the SlowGrowth component. This will cause the notification to appear at its previously specified initial size, and to start growing at the given growth rate. If the user interacts with the window, the notification will behave as described in Chapter 4. Additionally, the developer has two options for programmatically causing the notification to disappear. The `reset` method of the SlowGrowth component causes the notification to return to its initial size and resume growing. The `stop` method causes the notification to disappear, behaving as if it were dismissed by the user. This allows developers to dismiss or reset notifications that are no longer relevant to the application's behavior.

Appendix A provides a more detailed look at the API developed for SGL.

5.2.1 Sliding Window

One of the major challenges in implementing the SGL prototype was preventing *flicker* or *flashing* in the notifications as they scale. The original version of SGL featured a simple design for the notification itself: the SlowGrowth component started out as a JDialog of zero extent, and resized itself each time in order to produce the growth of the notification window. Unfortunately, the behavior of the repainting system in Java caused undesirable effects. Whenever the window resized, a visible "flash" would appear in the notification: the entire notification would briefly flash white before repainting with the correct content. This flash violated the principle of subtlety, as it caused a large amount of change on the screen. This tended to attract the user's attention, thus destroying the effectiveness of the slow-growth notification.

To solve this problem, a novel scheme was developed. This system is called the *sliding window* technique, and it solves the problem by *moving* the SlowGrowth component rather than resizing it. Figure 5-1 illustrates how the sliding window technique works. The SlowGrowth component remains at a fixed size and moves *inward* from

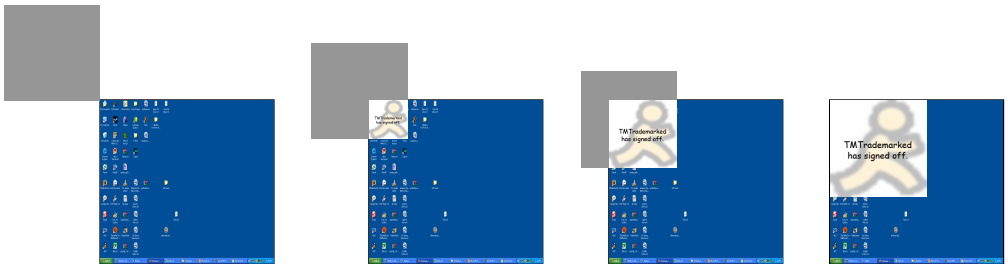

Figure 5-1: A visualization of the sliding window effect used in SGL

the corner of the screen. The NotifyImage component is placed within the bounds of the SlowGrowth component, and as the SlowGrowth container moves inward, the NotifyImage moves *outward* and/or is resized.[1] The net effect of this combination of motion and resizing the NotifyImage is that the SlowGrowth component is entirely invisible to the user, allowing only the NotifyImage to be displayed. This is because as the SlowGrowth component encroaches on the screen, the NotifyImage is moved and resized such that it covers exactly that portion of the SlowGrowth component which is inside the bounds of the user's screen.

The SlowGrowth component is initialized to the maximum size, and its location is set to a specified point called the *hidden point*. The hidden point defaults to `(-600,-600)`, which ensures that the SlowGrowth component is located entirely off of the visible screen. Developers are able to set the value of the hidden point so that it is appropriate for their own applications.

When the SlowGrowth component is instantiated, an array of starting locations is specified. Each starting location includes the following information: a start point for the SlowGrowth component (relative to the top left corner of the screen), a start point for the NotifyImage (relative to the top left corner of the SlowGrowth component), an *inward* vector specifying the direction of motion for the SlowGrowth component,

[1]Notice that if the SlowGrowth component is in the bottom right corner of the screen, the NotifyImage simply needs to resize and not move. This occurs because the NotifyImage initially appears in the top left corner of the SlowGrowth component, which is the corner that is moving inward. Thus, the NotifyImage remains stationary and simply resizes. In all other corners, though, the NotifyImage both resizes and moves.

and an *outward* vector specifying the direction of motion for the NotifyImage component. These parameters are currently hard-coded, allowing slow-growth notifications to appear solely at the corners of the screen. This restriction is unfortunate, and further work may be able to do away with it.

One problem with the sliding window system used in SGL is that it may interact poorly with multiple monitor environments. Because the SlowGrowth component is always full size, in multiple monitor environments the unused portions of the SlowGrowth component may be visible. Clearly, this is less than ideal. Future work is needed to examine ways to improve the sliding window technique, or eliminate it completely and replace it with other methods to achieve the same purpose.

5.3 NotifyImage

The NotifyImage component is responsible for handling the display of the notification's content. This component is what the user actually sees when they notice an SGL notification. A NotifyImage contains the components designed by the developer, and paints them to the screen in whichever of the three display modalities from Chapter 4 is selected. Every time the NotifyImage is painted, it defines a variable `scale`, which is defined to be the ratio of the NotifyImage's current width to its maximum width. (Note that currently SGL only supports *square* notifications. Thus basing the scale factor on the width is completely arbitrary, and would function equally well using height.) The canvas of the NotifyImage is then scaled symmetrically by this scale factor using the Java `Graphics2D` class. This scaling of the display surface allows developer to design notifications as if they were going to be displayed at full size, and ignore the subtleties caused by changing sizes. The `scale` method of `Graphics2D` automatically handles scaling the image appropriately to map from the original size to the new scaled version.

This basic scaling method is sufficient for the basic continuous visual zooming.

Supporting discontinuous semantic requires a very simple modification, however. In discontinuous semantic zooming, the notification may change dramatically at each distinct size. Thus, instead of storing one component, the NotifyImage is capable of storing an *array* of components, one for each of the size thresholds specified by the developer. Then, on every repaint, the current size of the notification is compared against the size thresholds specified by the developer. Once the appropriate threshold is determined, the NotifyImage selects that particular component from its array, and paints it onto the scaled graphics canvas.

This captures two of the three desired display modalities. Implementing continuous semantic zooming, however, requires significantly more complexity. Somehow, the NotifyImage needs to encapsulate information about the display of the notification at each particular state, and smoothly animate the transitions between states. To achieve this, SGL implements a *key-framing* system, where each key frame describes the state of the NotifyImage at particular instant in time.

5.3.1 Key-Framing in SGL

Key-framing is a well known concept from the field of computer animation. The idea in animation is to capture the state of the model in certain critical positions, known as key frames. When the animation is rendered, the computer interpolates the positions of the model in between the key frames such that the end result looks like a smooth motion. Applying this concept to SGL, the NotifyImage stores a series of key frames which capture the state of the notification.

Key frames are indexed by a single variable called *index*, which is always between 0.0 and 1.0, inclusive. This value represents the window's current size as a percentage of the maximum size. Every continuous semantic NotifyImage is required to have at least two key frames (indexed 0.0 and 1.0), representing the initial and final states of the notification. Aside from the index, each key frame contains an array of CompParam structures. These structures combine a JComponent with parameter

information representing its state. These five parameters (`x`, `y`, `w`, `h`, and `a`) allow the developer to specify the component's position within the NotifyImage (`x` and `y` represent this position in pixels relative to the top left corner of the NotifyImage), its dimensions (`w` and `h` are passed in as scale factors of the component's natural size), and its alpha value (`a` is a double ranging from 0.0 to 1.0, specifying the opacity of the component). Thus, developers can create continuous semantic notifications that feature various components moving around, resizing, and fading in or out.

On every repaint, the NotifyImage computes the `scale` variable as described previously. This scale factor can be compared to the index of the key frames stored within the NotifyImage. If the NotifyImage is at its maximum size, the data from the key frame with index 1.0 is drawn directly to the canvas with no further modification. If the NotifyImage is at some other size, it uses the scale factor to determine which two key frames it is between. The frame with the lower index is called `current` while the higher indexed frame is called `next`. The NotifyImage then defines a parameter α, where α represents the amount of progress the NotifyImage has made from `current` to `next`. As an example, if the `current` key frame has an index of 0.5, the `next` frame has an index of 1.0, and `scale` is currently 0.75, the value of the α parameter is 0.5, since the NotifyImage is halfway between the two key frames.

Once the value of α has been determined, the NotifyImage computes blended parameters for each component in the notification. In order to correctly update the parameters such that a smooth interpolation between key frame states can be achieved, the blended parameters are computed using the following formula:

$$\texttt{new.val} = (1.0 - \alpha) * \texttt{current.val} + \alpha * \texttt{next.val}$$

This blending function assigns higher weight to the `current` frame if the current size is closer to its index, and higher weight to the `next` frame as `scale` approaches `next.index`. Because the function is continuous, the resulting interpolation produces

a smooth animation when played back at speed. Thus, this key-framing system allows SGL to support the continuous semantic display modality.

The way components are described in the key-framing system is slightly different from the continuous visual and discontinuous semantic cases. In those cases, the developer designs the entire notification ahead of time, and passes it to the Notify-Image as a *single* component (or, in the case of discontinuous semantic, an array of components with one component for each desired size). In the key-framing system, each key frame describes the position of each *individual* component within the notification. Thus, in using the key-framing system, the notification is constructed piece by piece from specified components, while in the other display modes the notification is constructed ahead of time and simply displayed.

Chapter 6

Evaluation

The previous chapters have described the motivation for developing slow-growth notification and the actual details of a prototype system called SGL. This chapter discusses the evaluation of the SGL system. In order to determine the effectiveness of slow-growth notifications as compared to popup notifications, two user studies were conducted, a lab study and a field study. The following sections describe the studies and present the results.

6.1 Lab Study

The hypothesis for the lab study was as follows: *slow-growth notifications will interrupt users at natural task breaks more frequently than popups will, leading to improved performance.* In order to evaluate the hypothesis, 7 users were recruited by advertising on campus. These users were all MIT students of varying ages and gender, all of whom spend significant time using computers on a daily basis. Table 6.1 summarizes the characteristics of the users.

Table 6.1: A summary of user characteristics

Parameter	User Characteristics
Age	$ave = 22.4, min = 18, max = 27$
Gender	$male = 4, female = 3$

6.1.1 Design

To evaluate the effectiveness of slow-growth notifications, users were asked to perform a foreground task while being periodically interrupted by both slow-growth and popup notifications. The interruptions occurred at random intervals. The task chosen for this lab study was the task of typing text. This task was chosen for several reasons. Firstly, every user in the study population was familiar with the task of typing, and spent several hours a day performing it. Secondly, typing features easily identifiable sub-task boundaries between words, sentences, and paragraphs. These factors made typing text the ideal task for the purposes of this lab study.

In order to conduct the lab study, a custom application was constructed using the SGL toolkit. The bulk of the application consisted of two panels. The left panel was pre-populated with text, while the right panel was left blank initially. The left panel was referred to as the reference panel, and the right was called the user panel. The text in the reference panel consisted of 4 paragraphs of five sentences each. The sentences were randomly selected from a corpus consisting of the 720 so-called "Harvard Sentences" [1]. These sentences were originally designed to test voice quality in telecommunications. The reason they were chosen for this experiment is that all of the sentences are very similar in terms of several important metrics, such as number of words, number of characters per word, average word difficulty, and reading ease. This means that randomly combining sentences from this corpus produces reference text of consistent length and difficulty, and allows large quantities of such reference text to be generated simply. Using randomly generated reference text is preferable to using known passages of text, as it avoids any possible distortion of the data by users with prior knowledge of the texts. Since the reference text is being randomly

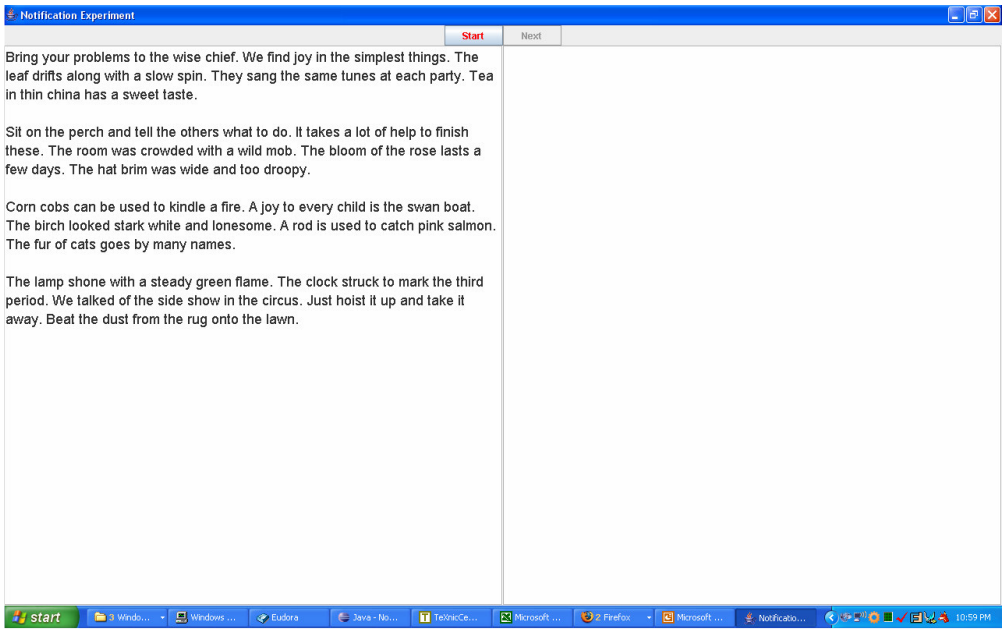

Figure 6-1: The lab study test application, as it appears on start up

generated, it is highly unlikely that users will have any prior knowledge of the text they are to transcribe. This means that they must read it for the first time while performing the study, which increases the cognitive load of the task.

Figure 6-1 shows what the application looked like on initial start up. This is what was presented to the users. All trials were conducted on the same computer (a laptop with a 13.3", 1280x800 screen) in the same physical location to minimize situational variance between trials.

Users were presented with the application and asked to transcribe the text from the left panel into the right as quickly and accurately as possible. When they finished one page, they were told to press the "Next" button on top of the application to proceed to the next page. Users were asked to complete as many pages of text as they could in 15 minutes. They were then given a short break to rest, and asked to complete another 15 minutes worth of work. The two data sets differed only in which type of notification would be used to interrupt the user: popup or slow-growth. For each of the two 15 minute data sets, the first page of data collected was thrown out.

This allowed users to become accustomed to each type of notification, and avoided the possibility of learning effects influencing the data.

During the process of transcribing the reference text, users were randomly interrupted with either popup or slow-growth notifications. Within each 15 minute set, only one type of notification (popup or slow-growth) was used. The order in which users saw these notifications was balanced, such that four users saw slow-growth for their first set and three saw popup for their first set. The slow-growth notifications also used varying growth rates, randomly choosing either 1, 5, or 10 pixels per second. All notifications would randomly appear at one of the four corners of the screen. Users were instructed to dismiss the notification by pressing the F2 key as soon as they noticed the window. A keyboard dismissal method was chosen over the mouse-based dismissal described in Chapter 4 in order to reduce homing lag. Since the task for this lab study was primarily keyboard based, asking the user to swith devices to use the mouse would have added an additional lag into the data. This would have made analyzing the cost of interruption more difficult.

The content of the notifications was always the same image, and users were instructed to ignore the content as it was not relevant to their task. This allowed the experiment to measure the base cost of the interruption *itself* by isolating the cost of the interruption from the cost of dealing with the actual content of the notification.

For each notification that the user dismissed, several statistics were recorded, including the response time, resume time, interruption point, page completion time, location on the screen (i.e., which corner the window appeared in), growth rate (for slow-growth notifications), type (popup or slow-growth), and final window size. Samples of the data collected and the format used are provided in Appendix B.

The *response time* was measured as the time difference in milliseconds between when the notification first appeared on screen and when the user pressed the F2 key to dismiss the notification. This measurement indicates how long it took for the user to notice the window.

Resume time was measured as the time difference in milliseconds between when the user pressed the F2 key and when they pressed any other key, resuming the typing task. The resume time is perhaps the most effective direct measure of interruption cost, since it measures how long it took for the user to find their previous place and continue with their task after being interrupted.

The *interruption point* was measured by recording the text the user had successfully transcribed when they dismissed the notification, and comparing the user's text to the reference text. For the task of transcribing text, four possible interruption points were identified. A notification could interrupt a user in the *middle* of a word, at the *end* of a word, at the end of a *sentence*, or at the end of a *paragraph*. Interruption in the middle of a word represents an interruption in the middle of a task, while interruption at any of the other three represents interruption at a task break. With respect to the task of typing text, there appear to be three identifiable subtasks: typing a word, typing a sentence, and typing a paragraph. Interruptions in the middle of a word are clearly in the middle of at least one of these subtasks, while interruptions at the ends of words, sentences, or paragraphs are at the boundaries of at least one of these subtasks. Thus, this statistic provided a useful measure for testing the hypothesis that slow-growth notifications would interrupt at task boundaries more frequently than popup notifications.

Finally, after the users completed the two 15 minute sets, they were given a subjective post-test questionnaire. This post-test interview sought to learn the user's subjective opinions on the differences between popup and slow-growth notification. Users were asked to rate the annoyance of both methods on a scale of 1 through 5, as well as rating how easily they were able to resume their task after being interrupted.

6.1.2 Results

After all users had completed their data sets, the results were collected and analyzed. In particular, there were five measurements that were most of interest in attempting

Table 6.2: Occurrences of each type of notification

Popup	Slow-Growth	Slow1	Slow5	Slow10
151	135	33	59	43

to study the benefits of slow-growth notification. These five measurements included the response time, the resume time, the interruption points, the page completion time, and the subjective responses collected from the users. The results and analysis for each of these measurements are presented in more detail in the subsections that follow. Note that in the graphs that follow, the three different growth rates of slow-growth tested may be presented individually, as well as in aggregate form. Therefore, the label *Slow-Growth* denotes the combined data, while the labels *Slow10*, *Slow5*, and *Slow1* refer to slow-growth notifications with growth rates of 10, 5, or 1 pixel per second, respectively. Table 6.2 shows how many of each type of notification were observed.

Response Time

The response time was measured as the time in milliseconds between the initial appearance of the notification and the user's dismissal by pressing F2. Since users were instructed to dismiss windows as soon as they noticed them, this provides a reasonable measure of how long it took users to notice the different classes of notification. Figure 6-2 shows the mean response time for slow-growth and popup notifications, and Figure 6-3 includes the mean response times for the three different growth rates of slow-growth.

The data here is interesting, but hardly surprising. Since the popup notifications appear instantly and change a large number of pixels, it should be expected that users would notice these notifications very rapidly (μ = 1613 ms). And similarly, since the slow-growth notifications appear slowly, it is unsurprising that users took significantly longer to notice these windows (μ = 18191 ms). Specifically, when the differing growth rates are considered individually, Slow10 took the least time to notice,

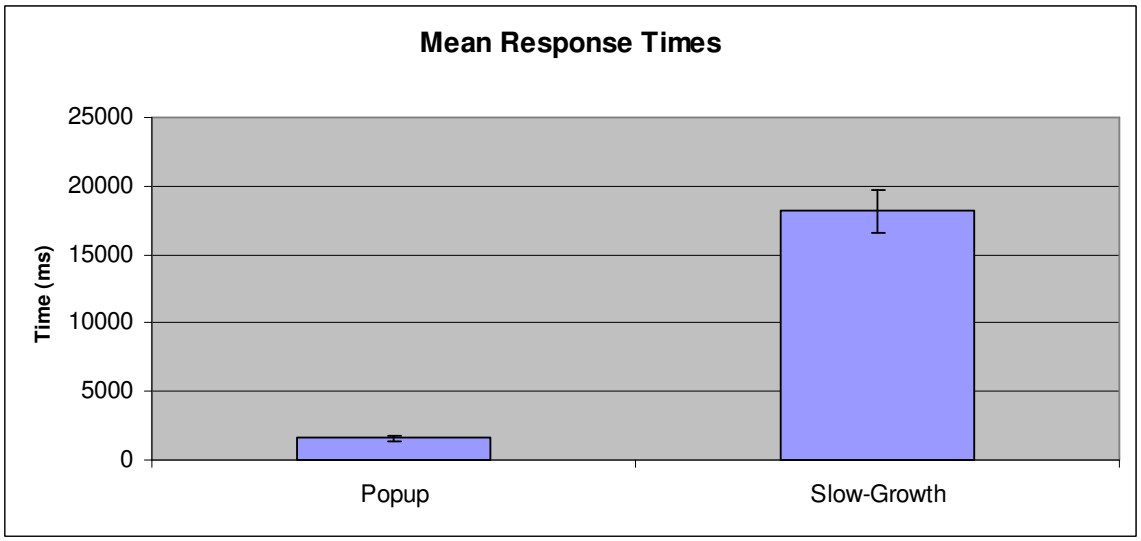

Figure 6-2: Mean response times for popup and aggregate slow-growth

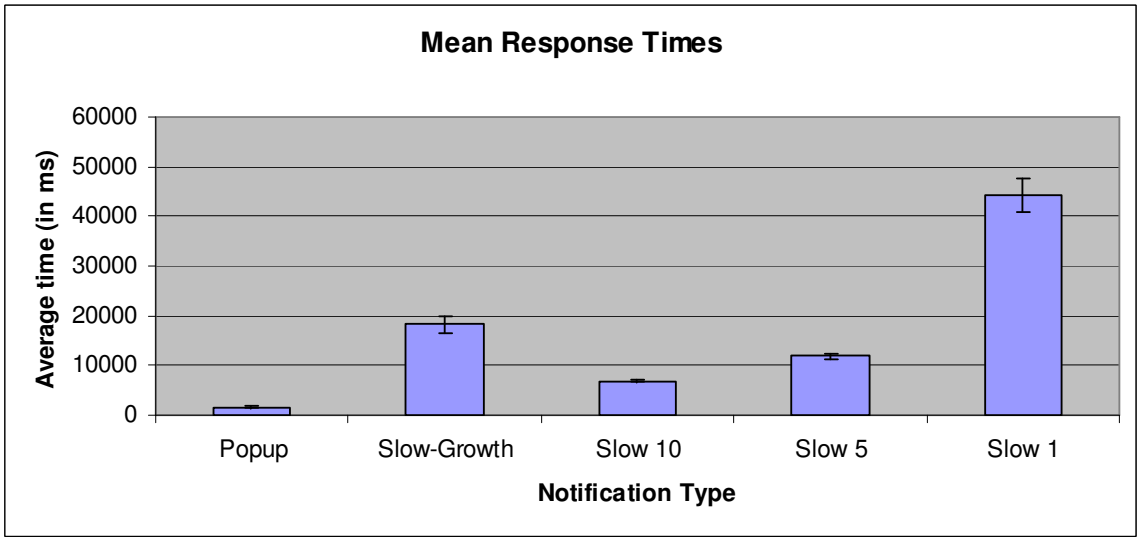

Figure 6-3: Mean response times including the differing growth rates

with Slow5 following, and Slow1 taking far and away the longest time to notice.

There is one particularly interesting characteristic of the data from the different growth rates. The data appears to indicate that each of these windows are noticed at different sizes (69, 59, and 44 pixels for Slow10, Slow5, and Slow1 respectively). However, it must be noted that these results were obtained by measuring the time delay between initial response and user action. Thus, these response times include any user processing delay, since there is some delay between noticing the window and acting to dismiss it. Since the three different classes were growing at different rates, assuming a constant reaction time of ρ seconds would reduce the actual window sizes at the moment of user attentional shift by of ρ, 5ρ, and 10ρ seconds for Slow1, Slow5, and Slow10 respectively. Therefore, it seems likely that all three types of notification were probably noticed at roughly the same window *size*. An eye-tracking system or similar solution would be needed to investigate this effect more precisely.

Resume Time

The resume time was measured as the delay between when the user pressed F2 to dismiss the notification and the next key press they enter. Because users were asked to complete their task as quickly as possible, this measurement provides a reasonable estimate of the amount of time it took for users to find their place and resume typing.

When the data was analyzed, there were 6 samples (2 slow-growth and 4 popup) out of 286 whose resume time was greater than 3 standard deviations away from the mean. These were considered outliers, and thus were removed from the rest of the analysis. Table 6.3 presents the data with and without the outliers to show the effect of removing these 6 samples. Figure 6-4 shows the mean resume time for both popup and slow-growth, and Figure 6-5 includes the mean resume times for each individual growth rate.

This set of data indicates several interesting results. Firstly, it indicates that users interrupted by slow-growth notifications required significantly less time to resume

Table 6.3: Mean resume times in milliseconds with standard deviations (including outliers, and with outliers removed)

	Outliers Included		Outliers Removed	
	μ	σ	μ	σ
Popup	1009	769	906	406
Slow-Growth	588	435	555	345
Slow10	534	358	534	358
Slow5	528	302	528	302
Slow1	767	646	638	397

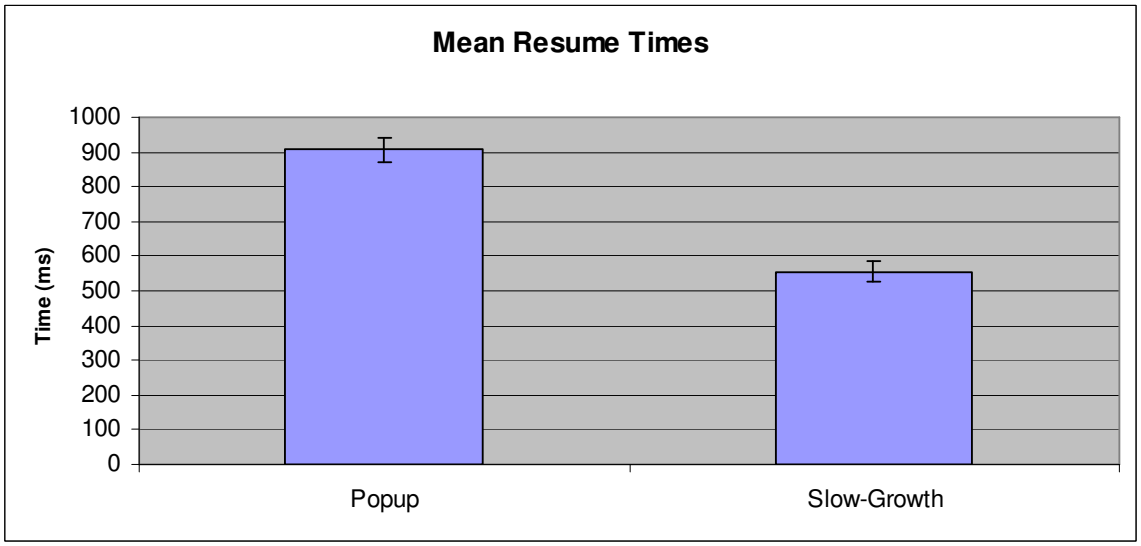

Figure 6-4: Mean resume times for popup and aggregate slow-growth

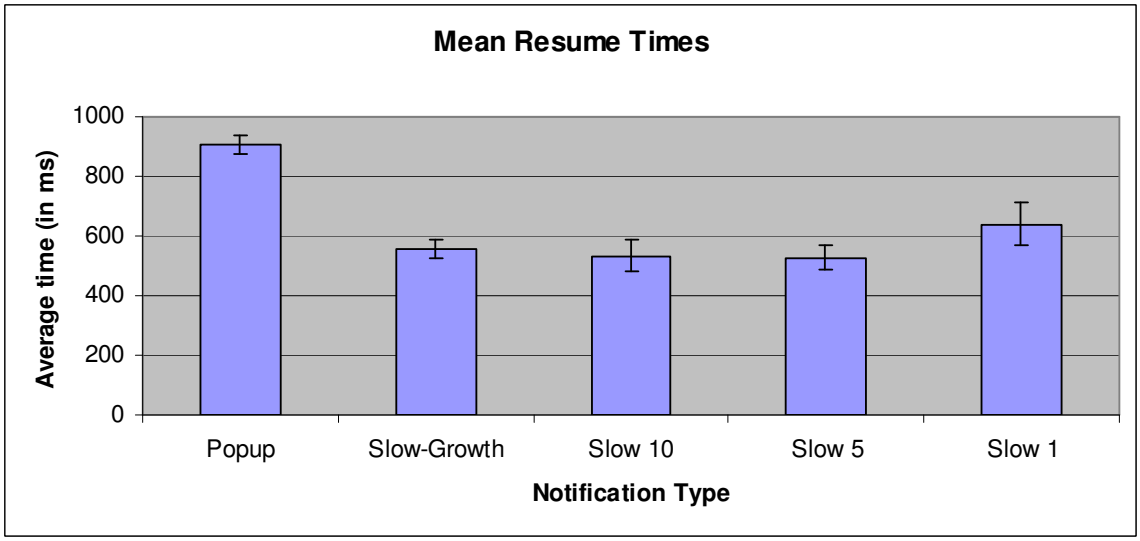

Figure 6-5: Mean resume times including differing growth rates

their tasks than users interrupted by popup notifications. As shown in Figure 6-4, the slow-growth notifications had approximately 39% lower resume times (555 ms as compared to 909 ms). This appears to satisfy the hypothesis that slow-growth notifications are significantly less disruptive than popup notifications to the user's ability to perform their primary task. (two-tailed t-test, $t_{284} = 7.746, p < 10^{-}12$)

Another interesting result is revealed by Figure 6-5. This graph shows the mean resume times for each individual growth rate. It was expected that the disruptiveness of a slow-growth window would be proportionate to its growth rate, with faster growing windows being more disruptive. However, the data appears to indicate otherwise. Given that the mean resume times for Slow10 and Slow5 are almost identical (534 ms and 528 ms, respectively), and that both are lower than the mean resume time for Slow1 (638 ms), it appears that the growth rate had little effect on the disruptiveness of the notification. A single-factor ANOVA test found no statistically significant difference. ($F(2, 130) = 1.179, p = 0.311$)

There are several reasons this might be so. First of all, the fact that the mean resume time for Slow1 appears higher than the mean resume time of the other growth rates may be a misleading data point. This is most likely due to the higher variance

in the Slow1 sample. The Slow1 sample contained only 31 samples, as opposed to 59 for Slow5 and 43 for Slow10. With more samples, the mean resume time for the Slow1 growth rate may be revealed to be lower. Another reason the data may not show a significant decrease in resume time across different growth rates may have to do with the design of the experiment, rather than a property of slow-growth notifications. For this experiment, the given task was transcribing text. In the model of this task, task breaks occur frequently. This means that there are plenty of moments for the user's attentional focus to widen and take in the notification. If task breaks were further apart, it is possible that the faster growing windows would disrupt users in the middle of tasks more frequently than the slower growing ones, leading to a difference in resume time. Additionally, transcribing text does not impose a particularly high cognitive load on the user. Thus, the user may be more able to resume their task quickly after a disruption than during a task involving a heavy cognitive load (i.e., video editing). Future experiments may help clarify this question.

Interruption Point

The interruption point was measured by comparing the reference text to the text the user had entered before they dismissed the notification. This measurement helps prove the central hypothesis of this thesis, that slow-growth notifications will be more successful at interrupting the user at task breaks than popup notifications.

First, the average distribution of possible interruption points was computed for each set of data, slow-growth and popup. The average number of characters, words, sentences, and paragraphs per page of data for each set was recorded. Table 6.4 presents this data. Using these numbers, the expected probabilities of interruption at each particular interruption point (mid-word, end of word, end of sentence, and end of paragraph) were computed. These probabilities represent what the data should look like if interruptions were truly random, based on the distribution of possible interruption points.

Table 6.4: Reference text characteristics (means per page of text)

	Characters	Words	Sentences	Paragraphs
Slow-Growth	809	159	20	4
Popup	816	160	20	4

Table 6.5: Number of interruptions at each possible interruption point

	Mid Word	End Word	End Sent.	End Para.
Slow-Growth	24	90	16	5
Popup	90	52	7	2

The actual data collected from the users is presented in Table 6.5. The actual percentages of the interruptions that occurred at each possible interruption point were then computed. Finally, these percentages were compared against the percentages expected if interruptions occurred randomly. Figure 6-6 shows this comparison.

The results here are encouraging. If interruptions were actually handled randomly, they would occur during the middle of a word most of the time. This is visible in the popup interruptions: nearly 60% of them occurred in the middle of a word. In contrast, slow-growth notifications only interrupted users in the middle of a word 18% of the time. This means that 82% of the time, slow-growth notifications interrupted the user at a natural task break, as compared to only 40% for popup notifications. The data clearly indicates that slow-growth notifications tend to be noticed at task breaks, as opposed to popup notifications, which are noticed almost randomly.

Note that even though the timing of the popup notifications was in fact random, there is still a difference between the distribution of popup interruptions and the expected random distribution. This is most likely due to an effect known as *chunking*: users tend to process tasks in conceptual chunks. In this case, this means that the user has already begun typing the word, and their fingers may well finish the word before responding to the interruption. Since the interruption points collected in this experiment measure what the user typed at the time they responded, this means that even though the popup appeared in the middle of the word, the data counted

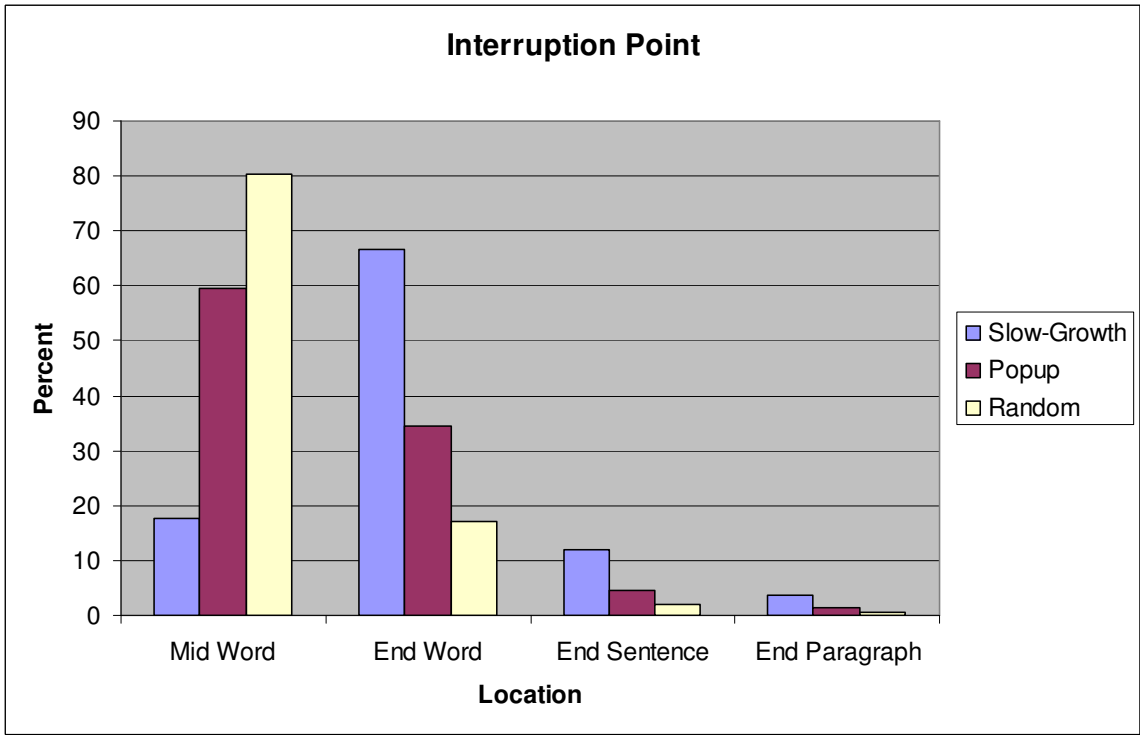

Figure 6-6: Interruption points

as an end of word interruption since the user was able to finish it before responding. However, this is acceptable, since it tends to bias results away from the hypothesis. Given this data, it appears that the hypothesis is largely correct, and that slow-growth notifications are able to interrupt at task breaks more often than popups, even without knowing any information about the user's task.

Page Completion Time

The page completion time was measured as the time delay in milliseconds from when the user pressed the "Start" button and began transcribing the text to when they pressed the "Next" button, indicating page completion. Figure 6-7 shows the results.

Notice that the average page completion time for pages with slow-growth notification was 166,469 ms, compared to 175,328 ms for popup notification pages. This represents a savings of 9 seconds over the course of 3 minutes, or approximately 5%.

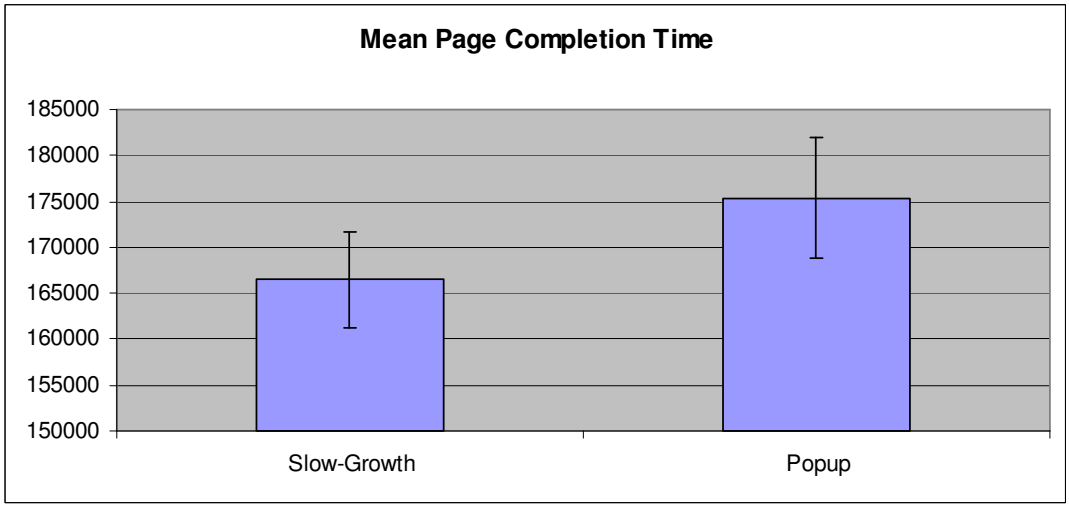

Figure 6-7: Mean page completion times for popup and aggregate slow-growth

While not a huge amount of improvement, this is still a promising figure. However, because the sample size is small, this result is not statistically significant. (two-tailed t-test, $t_{58} = 1.075, p = 0.29$)

Despite the marked improvements in resume time discussed above, slow-growth notification did not seem to make a significant difference in the overall task completion time. This is most likely because, even with popup notifications, the total task completion time is dominated by the time to actually perform the task, as opposed to the time to respond to the interruptions. Thus, the effect of using slow-growth notifications is swamped by the actual task itself. Future experiments may serve to amplify and clarify the exact extent of the time savings slow-growth notification can provide.

Subjective Responses

After a user completed their two 15 minute data sets, they were given a subjective post-test questionnaire. A copy of this questionnaire is provided in Appendix C. This post-test asked users to rate the annoyance of both popup and slow-growth notifica-

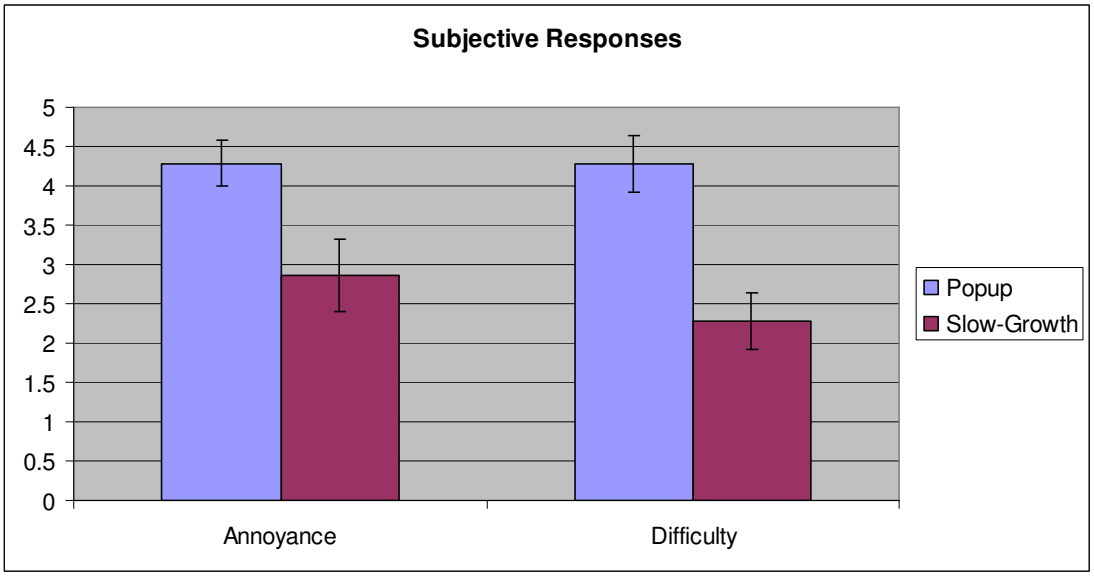

Figure 6-8: Subjective responses

tions on a scale of 1 through 5, with 1 being no annoyance and 5 being extremely annoying. They were also asked to rate, on a scale of 1 through 5, how difficult it was to resume their task after being interrupted by both popup and slow-growth notifications, with 1 being easy and 5 being hard. While the previous measurements have provided objective data showing the benefits of slow-growth notification, this measurement makes it possible to assess the subjective effects of notification type. Figure 6-8 illustrates the findings from the subjective questionnaire.

These results appear to confirm the hypothesis that slow-growth notifications are both less annoying and less disruptive than popup notifications. The data indicates that users found slow-growth notifications less annoying by approximately 33% (two-tailed t-test, $t_{12} = 2.64, p = 0.022$), and found it easier to resume their task by about 47%. (two-tailed t-test, $t_{12} = 3.93, p = 0.002$) As indicated by two-tailed t-tests, these results are statistically significant. Thus, the subjective results confirm that using slow-growth notifications may substantially improve the user experience and reduce user frustration.

6.2 Field Study

In order to evaluate the effectiveness of slow-growth notification under real world conditions, an informal field study was conducted. The users for this study consisted of five volunteer members of the User Interface Design group at MIT. Users were asked to run the field study application for one week, at which point their data was collected and analyzed. The following sections describe the design and results of this field study.

6.2.1 Design

For the field study, a small application was constructed. The application runs in the background, and randomly displays slow-growth notifications at approximately 8.5 minute intervals. Users were instructed to click on the notifications as soon as they noticed the window in order to dismiss them. Statistics were recorded after the user dismissed the notification. In particular, the field study was concerned with the response time and the average size of the notification when it was dismissed.

It was observed that all of the samples which attained maximum size had significantly higher response times than would be expected. Since the notifications stopped growing when they reached maximum size, there is a simple explanation for these higher response times. These samples appear to indicate that the notification appeared just as the user physically left their computer for an extended time, or as they hibernated their laptop. In other words, the notification would sit on screen without a user to actually observe it. When the user returned to their computer, they would then be able to dismiss the window. However, this does not accurately reflect when the user would have noticed the window under normal circumstances. Thus, all samples which attained maximum size were discarded as being irrelevant to the purpose of the study.

While the lab study used only continuous semantic notifications, the field study at-

Table 6.6: Occurrences of each notification type (growth rate and display mode) in the field study data

Total	Slow1	Slow5	Slow10	CV	DS	CS
535	188	183	164	181	191	163

tempted to study any performance differences between all three of the display modalities supported (continuous visual, discontinuous semantic, and continuous semantic). Additionally, the field study sought to investigate any performance difference between the three different growth rates tested in the lab study (1, 5, and 10 pixels per second). The field study was primarily concerned with the impact these two factors had how quickly users noticed the notification. Therefore, the notifications in the field study used one of the three display modalities chosen randomly, and one of the three growth rates used in the lab study. The figures that follow use *CV* to refer to continuous visual zooming, *DS* for discontinuous semantic zooming, and *CS* for continuous semantic zooming. Table 6.6 shows the number of occurrences for each growth rate and each display mode.

6.2.2 Results

The results from the field study were collected, and the average response time and average window sizes were recorded for each of the different display modalities supported. The hypothesis for this statistic was that continuous semantic zooming would be the least distracting, followed by continuous visual zooming, with discontinuous semantic zooming being the most distracting. Thus, the expectation was that continuous semantic zooming would have the highest response time and largest average window sizes and discontinuous semantic zooming would have the lowest. Figure 6-9 shows the average response time for each display mode, and Figure 6-10 shows the average notification size.

It is interesting that there does not appear to be a significant difference in either statistic across the three display modes. Neither the response times (single-

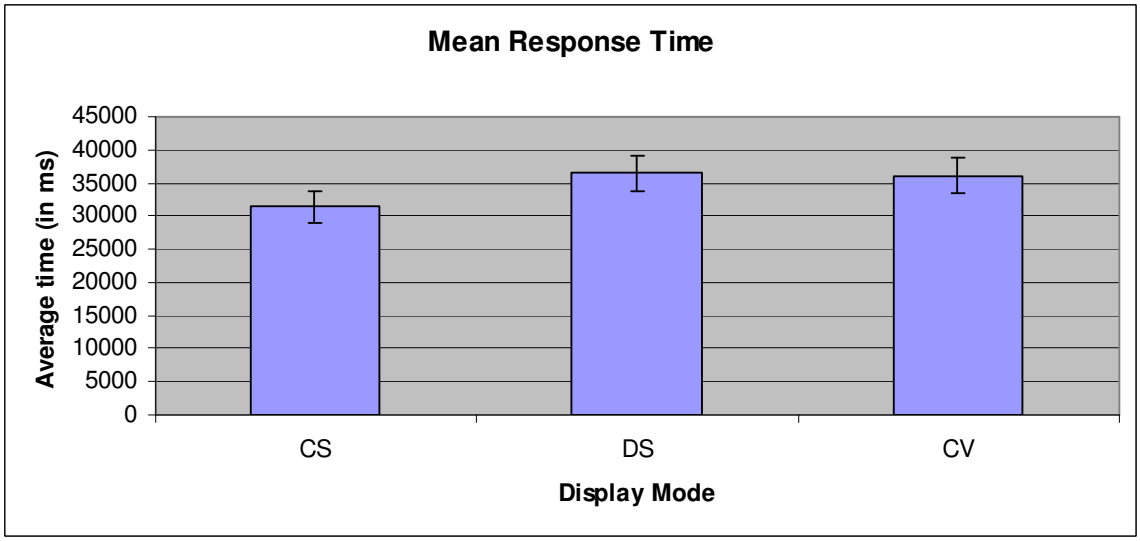

Figure 6-9: Average response times for each different display mode. CS is continuous semantic, DS is discontinuous semantic, and CV is continuous visual.

factor ANOVA, $F(2, 532) = 1.10, p = 0.334$) nor the notification sizes (single-factor ANOVA, $F(2, 532) = 0.525, p = 0.592$) showed any significant difference across display modalities. This indicates that the display modality did not appear to impact how disruptive the notification was to the user.

The other factor that the field study examined was the effect of differing growth rates. Using the same growth rates as in the lab study, the field study application displayed notifications that grew at either 1, 5, or 10 pixels per second. The hypothesis for this statistic was that a slower growth rate would be less disruptive than faster growth rates. Thus, the expectation was that Slow1 would have the highest response time and window size, with Slow10 having the lowest. Figure 6-11 shows the average response time for each of the different growth rates tested in the field study, and Figure 6-12 shows the average notification sizes

Comparing these results to the results presented in Figure 6-3 is interesting. Looking at the response times, Slow1 clearly had the largest response time. However, examining the average window sizes reveals something unexpected: Slow1 had the

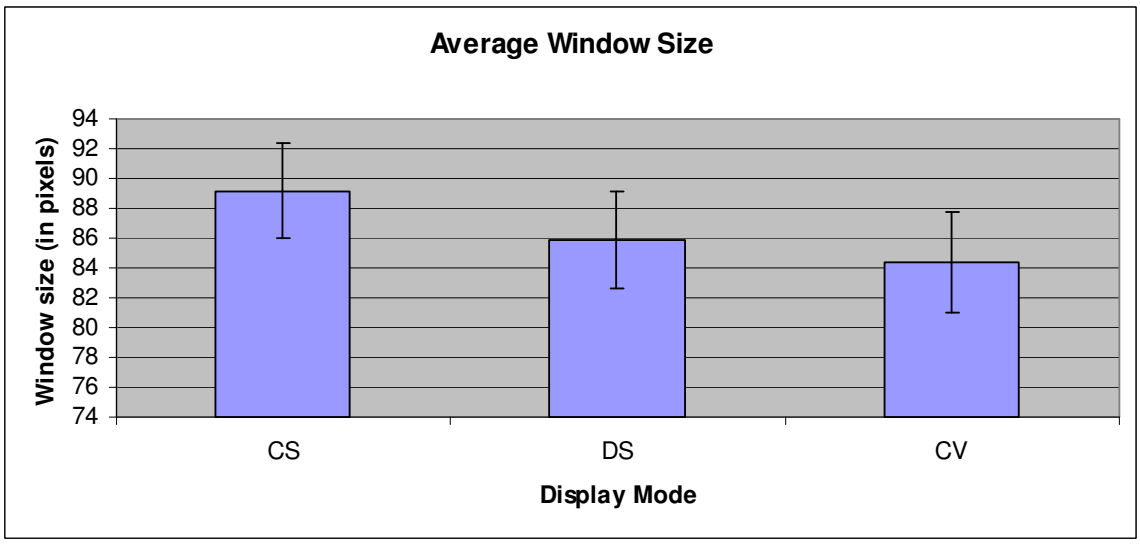

Figure 6-10: Average window for each different display mode

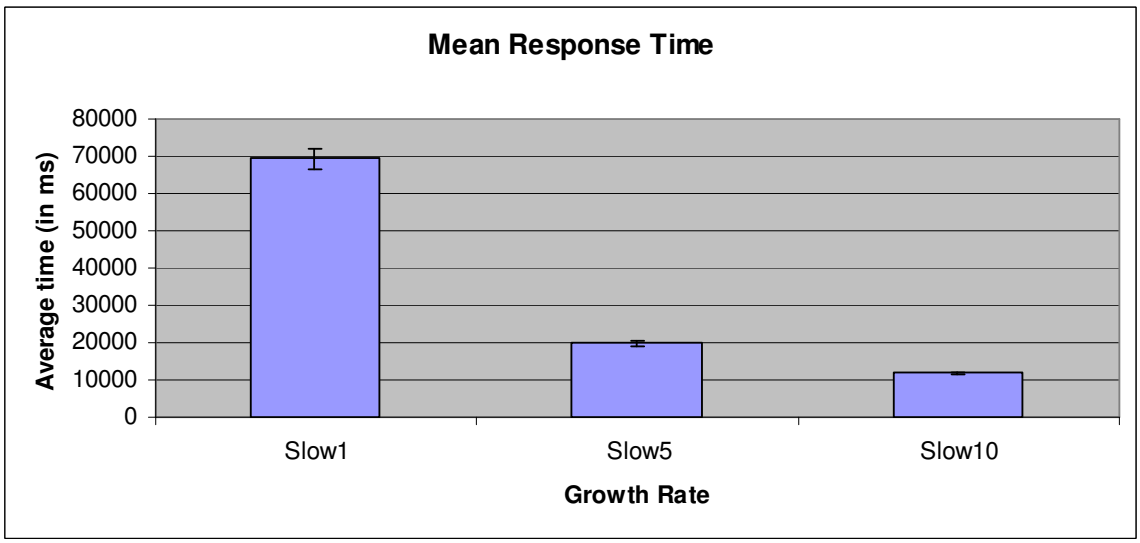

Figure 6-11: Average response times for each different growth rate

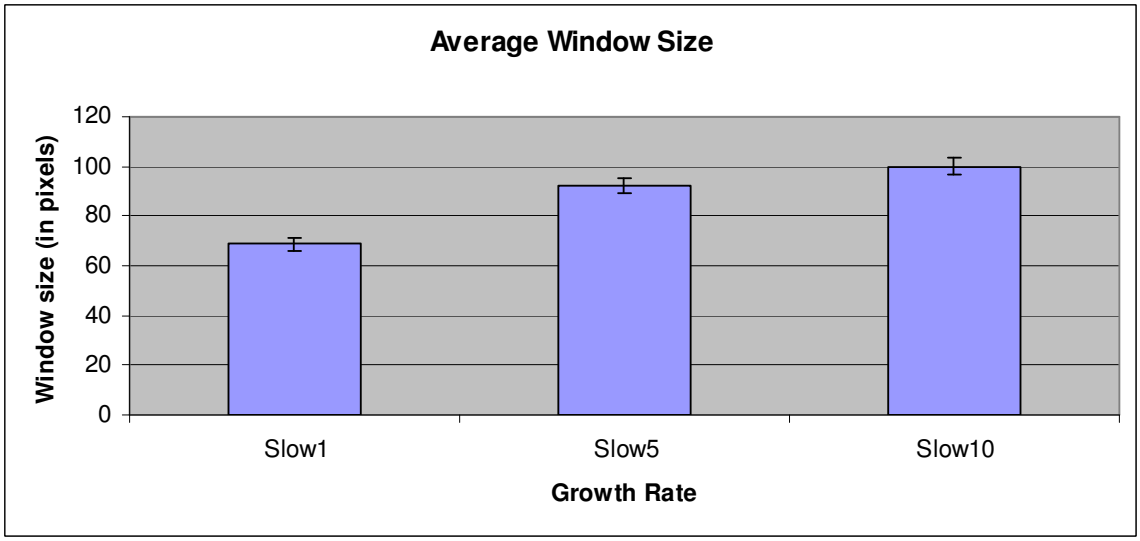

Figure 6-12: Average window for each different growth rate

smallest window size out of the three, while Slow5 and Slow10 had similar sizes. As mentioned previously, it seems likely that the windows were probably noticed at similar sizes, and that the data is confounded by a reaction delay: the measurement for response time used in this study includes the user's time to *act* on the notification in addition to their time to *notice* it. In addition to the delay introduced by the physical act of moving the mouse over to the corner, there's an additional cognitive processing delay. Even once the user has *seen* the window, there is some processing that must be done before they fully realize what action they have to take. In order to really gain a useful understanding of this statistic, these confounders must be removed. Experiments with eye tracking systems or with richer user interaction logs might help to clarify these results.

These results appear to indicate that the *size* of the notification is a better predictor for how soon it will be noticed than the amount of *motion* the content of the window undergoes. This is an interesting result, and may be worth further investigation. There are several intriguing experiments suggested by this result. For example, if size is truly the most important predictor, perhaps comparing slow-growth notifications

to popups of varying size would yield useful results. Similarly, testing the different display modalities against windows with much more content motion (e.g., animations, rapid flashing, or other sudden changes) could reveal interesting information. Other suggestions for future work along these lines are discussed in Section 7.2.2.

Chapter 7

Conclusion

This thesis set out to establish the principle of gradual awareness notification, and to prove the following hypothesis: **Gradual awareness notifications will be consumed by users at natural task breaks, improving performance and reducing annoyance when compared to traditional hard notification schemes.** In the course of examining this hypothesis, it has made several useful contributions to the understanding of notification. These contributions are discussed in detail below. The results presented in this thesis clearly indicate that gradual awareness notification is an interesting concept that merits additional study. The remaining section discusses future directions to carry this work in, and attempts to identify areas needing further investigation.

7.1 Contributions

This thesis has presented the concept of gradual awareness notification, and attempted to outline the general characteristics of such a notification system. The gradual awareness principle, where notifications begin as unnoticeable signals and gradually intensify, was laid forth. The principles presented here represent an alternate conceptualization of notification: this thesis presents notifications as attentional *requests*

rather than attentional *demands*. As more and more systems begin to require attention from users, it is essential that the model of notification shift from demand to request. Otherwise, users run the risk of severe information overload and collapse under the burden of competing attentional demands.

Along with introducing the principles of gradual awareness notification, this thesis attempted to place gradual awareness notification in its appropriate context by presenting a taxonomy of notification. Using this classification, it defined the types of problems and situations that can benefit most from gradual awareness techniques (status notifications and future alerts), as well as identifying areas where gradual awareness may not be the appropriate tactic (synchronous notifications and immediate alerts).

In addition to defining the underlying principle of gradual awareness notification, this thesis presented several design principles which are essential to the successful deployment of gradual awareness systems. First, gradual awareness notifications should be *subtle*, so as to avoid interrupting primary tasks at inopportune moments. Second, gradual awareness notifications should be *informative*, so that users can glean as much information as possible from short periods of attention. And third, gradual awareness notifications should be *efficient* to interact with, so that users can return to their primary tasks with a minimum of disruption.

Additionally, this thesis has also presented the implementation of a specific gradual awareness notification scheme for the desktop environment which relies on slowly growing windows. This method was referred to as slow-growth notification, and implemented in a particular toolkit called the Slow-Growth Library (SGL). The specific design of the SGL toolkit was presented, discussing design choices necessary to adhere to the design principles laid out above. The implementation details of SGL, including the innovative sliding window system, were presented to illustrate the challenges of implementing a gradual awareness system.

Finally, this thesis presented quantitative data demonstrating the benefits of slow-

growth notification over traditional popup-based notification systems by conducting a user study. The results were overwhelmingly positive. Slow-growth notifications appeared to be approximately 39% less disruptive than popup notifications, as measured by the difference in resume times. Subjectively, users found slow-growth notifications 47% less difficult to resume after, and 33% less annoying than popups. In addition, the user study confirmed the hypothesis by showing the slow-growth notification did a significantly better job of interrupting users at boundaries than popup notification: roughly 82% of the slow-growth notifications occurred at task boundaries, as opposed to only 40% for popups.

This thesis has demonstrated that slow-growth notification can be an effective tool for increasing user productivity and decreasing annoyance levels. Most importantly, it has shown that these improvements in notification performance can be achieved *without* complicated prediction functions or prior knowledge of the user's task. By exploiting the user's own cognitive task model, slow-growth notification can achieve these significant performance benefits in a simple, flexible, and extensible manner.

7.2 Future Work

There are two major paths to take with this work in the future. Firstly, the particular implementation of the SGL and slow-growth notification could be improved in several ways. Secondly, the general principles of gradual awareness notification merit further investigation. The following sections cover each of these directions.

7.2.1 Slow-growth Implementation

The first major improvement that could be made to the implementation of the SGL system would be to allow notifications to appear at non-corner locations. Since the current system relies on the sliding window system, slow-growth notifications are currently restricted only to the corners of the screen. Future work to improve and extend

the sliding window system could remove this restriction. Additionally, the sliding window system was mainly a work-around for specific details of a Java implementation. With further work, a better method of implementing slow-growth windows may be found, eliminating the need for the sliding window system altogether.

Additional interface refinements to the SGL toolkit are another area where future work may be of use. Experimenting with different growth rates may reveal more optimal speeds. A particularly interesting idea is the concept of adaptive growth rates, where the window grows faster or slower depending on the changing urgency of its information. For example, as a deadline for a meeting approaches, the notification could begin to grow faster, hopefully making itself more noticeable. This area could provide for even more effective notifications, and may be a promising feature in the future. Another interesting improvement in the future would be to incorporate a gesture based interaction scheme. With further work, the SGL toolkit could be adapted to include a gesture based input scheme. This would make the interaction with slow-growth notifications even more efficient than the current interface.

Finally, it would be instructive to incorporate the SGL toolkit into a real application and measure the effects of slow-growth notifications in a real-world setting with actual users. Currently, work is underway to incorporate SGL into an AIM client, but as of this moment, no application actually uses slow-growth notifications. This would provide a richer set of data for analyzing the impacts of slow-growth notification on user performance and annoyance levels.

7.2.2 Gradual Awareness Investigation

The results presented in this thesis indicate that gradual awareness notification is an interesting topic, and worthy of future exploration. One particular area of gradual awareness notification that seems particularly worthy of further investigation is the growth rate. The results indicated that growth rates had little impact on the resume time. However, as noted in Chapter 6, there are many possible explanations for this.

It would be useful and interesting to investigate these explanations. Experiments with a wider variety of growth rates or different tasks with longer subtasks or higher cognitive loads may reveal interesting interactions between the growth rate of slow-growth notifications and their effectiveness as notifications.

The results from both the field and the lab study produced interesting results concerning the response time for notifications. While the results seemed to indicate that Slow1 windows were noticed sooner than Slow10 windows, this may be accounted for by the user's reaction time delay, as discussed in the previous chapter. Future experiments are needed in order to determine whether the results collected for response times are valid, or whether some interesting property of slow-growth notification is obscured within them. There remains the interesting question of whether the *size* of the window or the amount of *motion* it undergoes is a better indication of how soon it will be noticed. In other words, are faster-growing windows more or less distracting than slower windows? Additionally, there remains the question of whether the amount of change in the contents of the notification (ie, how much the notification window flashes) has an impact on the disruptiveness of the notification or not. Future experiments could help clarify this by comparing the current prototype with windows that undergo much greater change per unit time. This question of whether notifications with a higher "optical flux" correspond to a less subtle attentional request is interesting, and certainly worthy of further study.

Additionally, the results presented in this thesis have focused exclusively on slow-growth notifications, an application of gradual awareness notification in the visual channel. However, the gradual awareness principle can apply to other input channels as well, such as the audio or tactile channels. Future investigation into gradual awareness systems for these channels may yield interesting or useful results. For example, imagine constructing a chair that used gradually increasing vibrations to alert users to incoming email. Such a system may have many practical advantages, since the user does not need to be looking at the screen to receive the notification.

In particular, the construction of multi-channel gradual awareness systems is an interesting idea. The effects of interruption on an input channel which is different from the one being used for the primary task has not been particularly well studied. The interaction between different input channels may provide opportunities to design even less disruptive notification systems than slow-growth. As an example, consider the hypothetical "slow phone" described in Chapter 1, which uses both tactile and audio channels to notify the user of incoming calls. Such applications remain a fascinating area for future experiments.

Appendix A

SGL API

This appendix documents the API for the SGL prototype, using the standard Javadoc format.

src

Class SlowGrowth

```
java.lang.Object
  └ src.SlowGrowth
```

```
public class SlowGrowth
extends java.lang.Object
```

SlowGrowth represents the container for a slow-growth notification. This class is the wrapper around the NotifyImage, and handles all of the interaction. In order to use a slow-growth notification, first instantiate an instance of this class. Then configure the parameters as appropriate, and invoke the start() method.

Field Summary

boolean	**popup** popup indicates whether the windows should instantaneously appear, or grow in.

Constructor Summary

SlowGrowth() Constructs a SlowGrowth notification in default state.

Method Summary

int	**getGrowthDelay**()
java.awt.Dimension	**getInitialSize**()
java.awt.Dimension	**getMaxSize**()
NotifyImage	**getNI**()
int	**getStartLocation**()
static void	**main**(java.lang.String[] args)
void	**reset**(int delay) reset() causes the notification to return to its initial size and to resume growth as

	normal.
void	**setGrowthDelay**(int d) Sets the growth delay of the current slow-growth notification.
void	**setInitialSize**(java.awt.Dimension i) Sets the initial size that this notification will appear at.
void	**setMaxSize**(java.awt.Dimension m) Sets the maximum size of this notification.
void	**setNI**(NotifyImage nin) Use this method to set a new NotifyImage.
void	**setStartLocation**(int s) Specifies a new starting location for this notification.
void	**start**() start() causes a SlowGrowth window to immediately display and start growing.
void	**stop**() stop() causes a SlowGrowth window to stop growing and reset itself to the hidden point.

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

popup

public boolean **popup**

popup indicates whether the windows should instantaneously appear, or grow in. Set popup to true to cause the windows to popup instantly.

Constructor Detail

SlowGrowth

public **SlowGrowth**()

Constructs a SlowGrowth notification in default state. Default state consists of the following: 200ms growth delay, 200x200 max size, starting location top left corner, this.ni set to CONT_VISUAL with no contents.

Method Detail

start

```
public void start()
```

start() causes a SlowGrowth window to immediately display and start growing.

---

stop

```
public void stop()
```

stop() causes a SlowGrowth window to stop growing and reset itself to the hidden point. Use stop when the notification is no longer needed.

---

main

```
public static void main(java.lang.String[] args)
```

---

setGrowthDelay

```
public void setGrowthDelay(int d)
```

Sets the growth delay of the current slow-growth notification. GROWTH_DELAY corresponds to how long the program should delay between resizing the notification in milliseconds. Thus, a growth delay of 1000 equates to a growth rate of 1 pixel per second. Higher growth delay equals slower notification growth.

Parameters:

`d` - - The desired growth rate

---

getGrowthDelay

```
public int getGrowthDelay()
```

Returns:

- The growth delay in milliseconds

---

setMaxSize

```
public void setMaxSize(java.awt.Dimension m)
```

Sets the maximum size of this notification. MAX_SIZE indicates how large the notification should be at its maximum extent.

Parameters:

`m` - - The desired new maximum size

---

getMaxSize

```
public java.awt.Dimension getMaxSize()
```

Returns:
- The maximum possible size of this notification

---

setInitialSize

```
public void setInitialSize(java.awt.Dimension i)
```

Sets the initial size that this notification will appear at. The default is initial size 0.

Parameters:
`i` - - The new initial size

---

getInitialSize

```
public java.awt.Dimension getInitialSize()
```

Returns:
- The initial size of this notification

---

setStartLocation

```
public void setStartLocation(int s)
```

Specifies a new starting location for this notification. startLocation indicates which corner the notification will appear in. 0 represents the top left corner, 1 is the top right, 2 is the bottom right, and 3 means the bottom left. Currently, only these four starting locations are supported.

Parameters:
`s` - - The new starting location for this notification

---

getStartLocation

```
public int getStartLocation()
```

Returns:
- The starting location of this notification, represented as an integer from 0 to 3.

---

setNI

```
public void setNI(NotifyImage nin)
```

Use this method to set a new NotifyImage.

Parameters:
`nin` - - The new NotifyImage desired

---

getNI

```
public NotifyImage getNI()
```

Returns:
- The NotifyImage of this notification

---

reset

```
public void reset(int delay)
```

reset() causes the notification to return to its initial size and to resume growth as normal. Use this when the notification needs to be reset but not dismissed.

---

src

Class NotifyImage

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └javax.swing.JComponent
              └src.NotifyImage
```

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable

---

```
public class NotifyImage
extends javax.swing.JComponent
```

NotifyImage handles displaying the actual content of the notification. The parameters here control how the notification is displayed, and what the actual content displayed at each tick is.

See Also:

Serialized Form

Nested Class Summary

static class	**NotifyImage.NoteMode** NoteModes represent the different display modalities allowed in the system.

Nested classes/interfaces inherited from class javax.swing.JComponent
javax.swing.JComponent.AccessibleJComponent

Field Summary

java.awt.Dimension	**max**

Fields inherited from class javax.swing.JComponent
TOOL_TIP_TEXT_KEY, UNDEFINED_CONDITION, WHEN_ANCESTOR_OF_FOCUSED_COMPONENT, WHEN_FOCUSED, WHEN_IN_FOCUSED_WINDOW

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

NotifyImage(java.awt.Dimension maxSize) Constructs a CONT_SEMANT NotifyImage with the specified max size and no contents.
NotifyImage(javax.swing.JComponent[] c, int[] s, java.awt.Dimension maxSize) Constructs a DISCONT_SEMANT NotifyImage using the specified components and size cutoffs.
NotifyImage(javax.swing.JComponent j, java.awt.Dimension maxSize) Constructs a CONT_VISUAL NotifyImage using the specified Component.

Method Summary

void	**addKeyFrame**(KeyFrame k) Adds a new KeyFrame to the NotifyImage.
NotifyImage.NoteMode	**getMode**() Returns the mode of this NotifyImage.
java.lang.String	**getModeString**() Returns the mode of this NotifyImage as a String.
int[]	**getSizes**() Returns the cutoff sizes for this NotifyImage.
void	**paintComponent**(java.awt.Graphics g)
void	**setMode**(NotifyImage.NoteMode m) Specifies the display mode for this NotifyImage.
void	**setSizes**(int[] s) Sets the cutoff sizes for DISCONT_SEMANT mode.

Methods inherited from class javax.swing.JComponent
addAncestorListener, addNotify, addVetoableChangeListener, computeVisibleRect, contains, createToolTip, disable, enable, firePropertyChange, firePropertyChange, firePropertyChange, getAccessibleContext, getActionForKeyStroke, getActionMap, getAlignmentX, getAlignmentY, getAncestorListeners, getAutoscrolls, getBorder, getBounds, getClientProperty, getComponentPopupMenu, getConditionForKeyStroke, getDebugGraphicsOptions, getDefaultLocale, getFontMetrics, getGraphics, getHeight, getInheritsPopupMenu, getInputMap, getInputMap, getInputVerifier, getInsets, getInsets, getListeners, getLocation, getMaximumSize, getMinimumSize, getNextFocusableComponent, getPopupLocation, getPreferredSize, getRegisteredKeyStrokes, getRootPane, getSize, getToolTipLocation, getToolTipText, getToolTipText, getTopLevelAncestor, getTransferHandler, getUIClassID, getVerifyInputWhenFocusTarget, getVetoableChangeListeners, getVisibleRect, getWidth, getX, getY, grabFocus, isDoubleBuffered, isLightweightComponent, isManagingFocus, isOpaque, isOptimizedDrawingEnabled, isPaintingTile, isRequestFocusEnabled, isValidateRoot, paint, paintImmediately, paintImmediately, print, printAll, putClientProperty, registerKeyboardAction, registerKeyboardAction, removeAncestorListener, removeNotify, removeVetoableChangeListener, repaint, repaint, requestDefaultFocus, requestFocus, requestFocus, requestFocusInWindow, resetKeyboardActions, reshape, revalidate, scrollRectToVisible, setActionMap, setAlignmentX, setAlignmentY, setAutoscrolls, setBackground, setBorder, setComponentPopupMenu, setDebugGraphicsOptions,

setDefaultLocale, setDoubleBuffered, setEnabled, setFocusTraversalKeys, setFont, setForeground, setInheritsPopupMenu, setInputMap, setInputVerifier, setMaximumSize, setMinimumSize, setNextFocusableComponent, setOpaque, setPreferredSize, setRequestFocusEnabled, setToolTipText, setTransferHandler, setVerifyInputWhenFocusTarget, setVisible, unregisterKeyboardAction, update, updateUI

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, addPropertyChangeListener, addPropertyChangeListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalKeys, getFocusTraversalPolicy, getLayout, getMousePosition, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, printComponents, remove, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusCycleRoot, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setLayout, transferFocusBackward, transferFocusDownCycle, validate

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, contains, createImage, createImage, createVolatileImage, createVolatileImage, dispatchEvent, enable, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getForeground, getGraphicsConfiguration, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputContext, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocale, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getToolkit, getTreeLock, gotFocus, handleEvent, hasFocus, hide, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isPreferredSizeSet, isShowing, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, postEvent, prepareImage, prepareImage, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, resize, resize, setBounds, setBounds, setComponentOrientation, setCursor, setDropTarget, setFocusable, setFocusTraversalKeysEnabled, setIgnoreRepaint, setLocale, setLocation, setLocation, setName, setSize, setSize, show, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

max

public java.awt.Dimension **max**

Constructor Detail

NotifyImage

```
public NotifyImage(java.awt.Dimension maxSize)
```

Constructs a CONT_SEMANT NotifyImage with the specified max size and no contents. In order for this to work, key frames must be provided subsequently.

Parameters:

`maxSize` - - the maximum notification size

NotifyImage

```
public NotifyImage(javax.swing.JComponent j,
                   java.awt.Dimension maxSize)
```

Constructs a CONT_VISUAL NotifyImage using the specified Component.

Parameters:

`j` - - JComponent to be displayed

`maxSize` - - Maximum notification size

NotifyImage

```
public NotifyImage(javax.swing.JComponent[] c,
                   int[] s,
                   java.awt.Dimension maxSize)
```

Constructs a DISCONT_SEMANT NotifyImage using the specified components and size cutoffs. Note that comps.size must be equal to sizes.size + 1.

Parameters:

`c` - - the JComponents to display

`s` - - The size thresholds

`maxSize` - - The maximum notification sizes

Method Detail

addKeyFrame

```
public void addKeyFrame(KeyFrame k)
```

Adds a new KeyFrame to the NotifyImage. KeyFrames are always stored in sorted order.

Parameters:

`k` - - new KeyFrame

---

setSizes

```
public void setSizes(int[] s)
```

Sets the cutoff sizes for DISCONT_SEMANT mode. The first value is the size at which the displayed component will switch from the first component of comps to second, and so forth.

Parameters:
s -

---

getSizes

```
public int[] getSizes()
```

Returns the cutoff sizes for this NotifyImage.

Returns:
an array representing cutoff sizes

---

setMode

```
public void setMode(NotifyImage.NoteMode m)
```

Specifies the display mode for this NotifyImage.

Parameters:
m - - The desired NoteMode (CONT_VISUAL, DISCONT_SEMANT, or CONT_SEMANT)

---

getMode

```
public NotifyImage.NoteMode getMode()
```

Returns the mode of this NotifyImage.

Returns:
mode

---

getModeString

```
public java.lang.String getModeString()
```

Returns the mode of this NotifyImage as a String. Useful for testing and debugging.

Returns:
String mode

---

paintComponent

```
public void paintComponent(java.awt.Graphics g)
```

Overrides:

`paintComponent` in class `javax.swing.JComponent`

---

src

Class KeyFrame

```
java.lang.Object
  └ src.KeyFrame
```

```
public class KeyFrame
extends java.lang.Object
```

A KeyFrame represents the state of the notification at a specified percentage of full size. Thus, a KeyFrame contains the following: An index, representing the percentage of full size described by this frame. A list of CompParams, each representing the state of a given component in this frame. Notes and limitations of KeyFrames: 1) The NotifyImage must always contain at least two KeyFrames: one for 0.0 and one for 1.0. 2) Each KeyFrame should contain all components that will be shown in the final frame. To hide these components, set the alpha to 0.0 3) All components should appear in the same order in each key frame.

Field Summary

java.util.ArrayList<CompParam>	**comps**
double	**index**

Constructor Summary

KeyFrame(double i) Constructs a new KeyFrame with the specified index i

Method Summary

void	**add**(CompParam cp) Adds a CompParam to this KeyFrame.

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

index

```
public double index
```

---

comps

```
public java.util.ArrayList<CompParam> comps
```

Constructor Detail

KeyFrame

```
public KeyFrame(double i)
```

Constructs a new KeyFrame with the specified index i

Parameters:

`i` - - the KeyFrame's index (must be between 0.0 and 1.0)

Method Detail

add

```
public void add(CompParam cp)
```

Adds a CompParam to this KeyFrame.

Parameters:

`cp` - - the CompParam to add.

---

src

Class CompParam

```
java.lang.Object
  └ src.CompParam
```

```
public class CompParam
extends java.lang.Object
```

A CompParam represents the state of a particular component at a particular instant in time. It contains the component itself, a value for opacity, a size, and a position. CompParams are used inside of KeyFrames.

Field Summary

double	**alpha**
double	**h**
javax.swing.JComponent	**label**
double	**w**
double	**x**
double	**y**

Constructor Summary

CompParam(CompParam cp) Constructs a copy of the given CompParam
CompParam(javax.swing.ImageIcon i) Constructs a new CompParam with a JLabel containing the specified ImageIcon.
CompParam(javax.swing.JComponent j) Constructs a new CompParam with the given JComponent.
CompParam(java.lang.String s) Constructs a new CompParam with a JLabel containing the specified String.
CompParam(java.net.URL u) Constructs a new CompParam with a JLabel with an image from the specified URL.

Method Summary

java.lang.Object	**clone**()
void	**setParams**(double a, double x, double y, double w, double h) Sets the parameters of the CompParam

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

label

public javax.swing.JComponent **label**

---

alpha

public double **alpha**

---

x

public double **x**

---

y

public double **y**

---

w

public double **w**

---

h

public double **h**

Constructor Detail

CompParam

public **CompParam**(java.lang.String s)

Constructs a new CompParam with a JLabel containing the specified String. Uses the default parameters of 0.0,0,0,1,1

Parameters:
`s` - - the text to be displayed.

CompParam

```
public CompParam(javax.swing.ImageIcon i)
```

Constructs a new CompParam with a JLabel containing the specified ImageIcon. Uses the default parameters of 0.0,0,0,1,1

Parameters:
`i` - - the ImageIcon to display

CompParam

```
public CompParam(java.net.URL u)
```

Constructs a new CompParam with a JLabel with an image from the specified URL. Uses the default parameters of 0.0,0,0,1,1

Parameters:
`u` - - the URL of the image to load

CompParam

```
public CompParam(javax.swing.JComponent j)
```

Constructs a new CompParam with the given JComponent. Uses the default parameters of 0.0,0,0,1,1

Parameters:
`j` - - the desired JComponent

CompParam

```
public CompParam(CompParam cp)
```

Constructs a copy of the given CompParam

Parameters:
`cp` - - the CompParam to be copied

Method Detail

setParams

```
public void setParams(double a,
                      double x,
                      double y,
                      double w,
                      double h)
```

Sets the parameters of the CompParam

Parameters:

`a` - - the desired alpha
`x` - - the desired x position
`y` - - the desired y position
`w` - - how much to scale the width of the component
`h` - - how much to scale the height of the component

---

clone

```
public java.lang.Object clone()
```

Overrides:

`clone` in class `java.lang.Object`

Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Appendix B

Sample Data and File Formats

This appendix contains samples of the data files recorded during the user study. There were two file formats used. First is the individual file, which represents the data recorded for one particular interruption. The second format is the page-level file, which represents the data recorded for all of the interruptions in a page. In particular, the page-level file is used to record the resume time of the notification.

```
<slowgrowth version='6'>
      <rectangle x='-141' y='-141' w='59' h='59'/>
      <screen sw='1280' sh='800'/>
      <time t='6828'/>
      <mode m='continuous semantic'/>
      <popup p='slow'/>
      <growth g='100'/>
      <reference_text>
            The urge to write short stories is rare. A rag will soak up
            spilled water. A steep trail is painful for our feet. Five
            years he lived with a shaggy dog. He takes the oath of
            office each March.

            A sullen smile gets few friends. The harder he tried the
            less he got done. Choose between the high road and the low.
            Just hoist it up and take it away. He wrote his last novel
            there at the inn.

            Their eyelids droop for want of sleep. The vamp of the shoe
            had a gold buckle. At that high level the air is pure. The
            jacket hung on the back of the wide chair. A Tusk is used
            to make costly gifts.

            Shape the clay gently into block form. The sofa cushion is
            red and of light weight. A gem in the rough needs work to
            polish. Watch the log float in the wide river. The pipe
            began to rust while new.
      </reference_text>
      <user_text>
            The urge to write short stories is rare. A rag will soak up
            spilled water. A steep trail is painful to our feet. Five
            years he lived with a shaggy dog. He takes the oath of
            office each March.

            A sullen smile gets few friends. The harder he tried the
            less he got done. Choose between the high road and the low.
            Just hoist it up and take it away.
      </user_text>
</slowgrowth>
```

```
<typetester>
      <page p='2' pop='slow'/>
      <time t='157985'/>
      <notification n='-384802578'>
      <type t='slow'/>
      <resume_time r='219'/>
      <growth g='1000'/>
      </notification>
      <notification n='-1150380687'>
      <type t='slow'/>
      <resume_time r='172'/>
      <growth g='100'/>
      </notification>
      <notification n='1580379828'>
      <type t='slow'/>
      <resume_time r='125'/>
      <growth g='100'/>
      </notification>
      <notification n='572480934'>
      <type t='slow'/>
      <resume_time r='78'/>
      <growth g='200'/>
      </notification>
      <reference_text>
            The urge to write short stories is rare. A rag will soak up spilled
            water. A steep trail is painful for our feet. Five years he lived
            with a shaggy dog. He takes the oath of office each March.

            A sullen smile gets few friends. The harder he tried the less he got
            done. Choose between the high road and the low. Just hoist it up and
            take it away. He wrote his last novel there at the inn.

            Their eyelids droop for want of sleep. The vamp of the shoe had a
            gold buckle. At that high level the air is pure. The jacket hung on
            the back of the wide chair. A Tusk is used to make costly gifts.

            Shape the clay gently into block form. The sofa cushion is red and
            of light weight. A gem in the rough needs work to polish. Watch the
            log float in the wide river. The pipe began to rust while new.
      </reference_text>
      <user_text>
            The urge to write short stories is rare. A rag will soak up spilled
            water. A steep trail is painful to our feet. Five years he lived
            with a shaggy dog. He takes the oath of office each March.

            A sullen smile gets few friends. The harder he tried the less he got
            done. Choose between the high road and the low. Just hoist it up and
            take it away. He wrote his last novel there at the inn.

            Their eyelids droop for want of sleep. The vamp of the shoe had a
            gold buckle. At that high level the air is pure. The jacket hung on
            the back of the wide chair. A Tusk is used to make costly gifts.

            Shape the clay gently into block form. The sofa cushion is red and
            of light weight. A gem in the rough needs work to polish. Watch the
            log float in the wide river. The pipe began to rust while new.
      </user_text>
      <text_stats words='161' chars='783'/>
</typetester>
```

Appendix C

Post-test Questionnaire

This appendix contains the post-test questionnaire that was given to all participants in the study.

POST-TEST QUESTIONNAIRE **User ID_____**

1. Circle the number that best represents **how annoying** you found each kind of notification:

	Not annoying		Somewhat annoying		Very annoying
Instant popups	1	2	3	4	5
Growing popups	1	2	3	4	5

2. Circle the number that best represents **how hard it was to find your place in the text** after being interrupted by each kind of notification:

	Not hard		Somewhat hard		Very hard
Instant popups	1	2	3	4	5
Growing popups	1	2	3	4	5

3. Circle the number that best represents how much you agree with the following statement: "**I found myself scanning the corners of the screen looking for notifications**":

Strongly disagree		Neutral		Strongly agree
1	2	3	4	5

4. To the best of your recollection, **how many interruptions per page** (on average) did you see during each part of the study?

	Average number of notifications per page
Instant popups	_____
Growing popups	_____

Bibliography

[1] Ieee recommended practice for speech quality measurements. *IEEE Transactions on Audio and Electroacoustics*, AU-17(3):225–246, September 1969.

[2] Dimitris Achlioptas and Eric Horvitz. Principles of bounded deferral for balancing information awareness with interruption. Technical Report MSR-TR-2005-87, Microsoft Research (MSR), July 2005.

[3] Piotr D. Adamczyk and Brian P. Bailey. If not now, when?: the effects of interruption at different moments within task execution. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 271–278, New York, NY, USA, 2004. ACM Press.

[4] B. Bailey, J. Konstan, and J. Carlis. Measuring the effects of interruptions on task performance in the user interface. In *IEEE Conference on Systems, Man, and Cybernetics 2000*, pages 757–762. IEEE, 2000.

[5] B. Bailey, J. Konstan, and J. Carlis. The effects of interruptions on task performance, annoyance, and anxiety in the user interface. In *Proceedings of INTERACT*, 2001.

[6] Brian P. Bailey and Shamsi T. Iqbal. Leveraging changes in mental workload during task execution to mitigate effects of interruption. Technical report, University of Illinois at Urbana-Champaign, August 2005.

[7] Brian P. Bailey and Joseph A. Konstan. On the need for attention-aware systems: Measuring effects of interruption on task performance error rate and affective state. *Computers in Human Behavior*, 22(4):685–708, July 2006.

[8] Benjamin B. Bederson and James D. Hollan. Pad++: a zooming graphical interface for exploring alternate interface physics. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 17–26, New York, NY, USA, 1994. ACM Press.

[9] James "Bo" Begole, Nicholas E. Matsakis, and John C. Tang. Lilsys: Sensing unavailability. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 511–514, New York, NY, USA, 2004. ACM Press.

[10] Adam Bodnar, Richard Corbett, and Dmitry Nekrasovski. Aroma: ambient awareness through olfaction in a messaging application. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, pages 183–190, New York, NY, USA, 2004. ACM Press.

[11] Marvin M. Chun and Jeremy M. Wolfe. Visual attention. In E. B. Goldstein, editor, *Blackwell's Handbook of Perception*, chapter 9, pages 272–310. Blackwell, Oxford, UK, July 2001.

[12] Mary Czerwinski, Edward Cutrell, and Eric Horvitz. Instant messaging and interruption: Influence of task type on performance. In *Proceedings of OZCHI 2000*, pages 356–361, 2000.

[13] Anton N. Dragunov, Thomas G. Dietterich, Kevin Johnsrude, Matthew McLaughlin, Lida Li, and Jonathan L. Herlocker. Tasktracer: a desktop environment to support multi-tasking knowledge workers. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 75–82, New York, NY, USA, 2005. ACM Press.

[14] James Fogarty, Scott E. Hudson, and Jennifer Lai. Examining the robustness of sensor-based statistical models of human interruptibility. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 207–214, New York, NY, USA, 2004. ACM Press.

[15] http://maps.google.com.

[16] Eric Horvitz and Johnson Apacible. Learning and reasoning about interruption. In *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*, pages 20–27, New York, NY, USA, 2003. ACM Press.

[17] Eric Horvitz, Andy Jacobs, and David Hovel. Attention-sensitive alerting. In *UAI*, pages 305–313, 1999.

[18] Eric Horvitz, Paul Koch, and Johnson Apacible. Busybody: creating and fielding personalized models of the cost of interruption. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 507–510, New York, NY, USA, 2004. ACM Press.

[19] Shamsi T. Iqbal. Mews-it: A mental workload based system for interruption timing. In *UIST Doctoral Symposium 2005*.

[20] Paul P. Maglio and Christopher S. Campbell. Tradeoffs in displaying peripheral information. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 241–248, New York, NY, USA, 2000. ACM Press.

[21] Stefan Marti and Chris Schmandt. Physical embodiments for mobile communication agents. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 231–240, New York, NY, USA, 2005. ACM Press.

[22] D. Scott McCrickard and C. M. Chewar. Attuning notification design to user goals and attention costs. *Commun. ACM*, 46(3):67–72, 2003.

[23] D. Scott McCrickard, C. M. Chewar, Jacob P. Somervell, and Ali Ndiwalana. A model for notification systems evaluation - assessing user goals for multitasking activity. *ACM Trans. Comput.-Hum. Interact.*, 10(4):312–338, 2003.

[24] Dean Rubine. Specifying gestures by example. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 329–337, New York, NY, USA, 1991. ACM Press.

[25] D. J. Simons and C. F. Chabris. Gorillas in our midst: Sustained inattentional blindness for dynamic events. *Perception*, 28(9):1059–1074, 1999.

[26] Maarten van Dantzich, Daniel Robbins, Eric Horvitz, and Mary Czerwinski. Scope: Providing awareness of multiple notifications at a glance. In *Advanced Visual Interfaces 2002*, May 2002.