Post-Autistic Software

Peter J. Denning[1] and John E. Hiles[2]

[1] Naval Postgraduate School, Monterey, CA USA, pjd@nps.edu
[2] Naval Postgraduate School, Monterey, CA USA, jehiles@nps.edu

Draft 5 -- 12/1/05

1 Introduction

One of the biggest challenges facing computer science is that software is autistic.

Medical autism is a terrible disease. Its main symptoms -- that the person is inward looking, uncommunicative, and anti-social -- cause untold grief for the families of its victims. We do not lightly make an analogy with software.

It is said that the main difficulty with software is that we have no design processes that reliably lead us to software that is dependable, reliable, usable, safe, and secure. These goals are complex and difficult to achieve. They are evaluated along two dimensions, which we call *engineering* and *social,* Success in the engineering dimension is evaluated by "correctness" -- that software satisfies rigorous specifications on its function and performance. Correctness is fiendishly difficult because most systems are complex, they cannot be specified with sufficient rigor, the proofs are intractable, and testing is inconclusive. Success in the social dimension is evaluated by "fitness" -- that software match the practices, expectations, intentions, ambitions, assessments, and experiences of users. Fitness is difficult because each individual evaluates according to personal interests and concerns.

Given that most of our rigorous tools and methods are part of the engineering dimension, it is little surprise that the most common complaints about software are social -- fragile, brittle, low quality, anti-social, and autistic (lacking context-awareness) are all good illustrations. We have few rigorous tools and methods for evaluating and designing software in the social dimension, where fitness rather than correctness is the dominant consideration.

Our objective in this paper is to propose a framework for understanding and measuring fitness, and a design process for achieving it. We believe this is the key to achieving a new, post-autistic generation of software. We will discuss these claims:

- Lack of context awareness makes software autistic.
- A new measure, fitness, is needed to assess software's social interactions in its domain of action.
- Virtual memory systems are a case study showing autism (thrashing) cured by context awareness from the principle of locality.
- Although unfit software is common, there are good examples of fit software. They exhibit five levels of fitness.
- Software (and its designers) can achieve fitness by using the locality principle to apprehend context.
- Designers of context-aware software must be aware of tradeoffs between certain privacy concerns and the value brought by context-awareness.

2 Lack of context awareness makes software autistic

Infantile Autism develops in children before 30 months of age. It is characterized by impairments in verbal and non-verbal communication, imagination, and social interaction. Autistic children are unable to develop normal relationships with others. They remain aloof and prefer a world of their own. They exhibit temper tantrums when required to change from a favored pattern to something else. When they do respond verbally they often repeat a question without answering it. Sometimes the people around an autistic child "learn" the abnormal behavior, adapting to unreasonable requirements of the child in order to avoid a tantrum. Autistic children often exhibit savant abilities in some narrow areas. The essential factor in autism is that its victims have lost their ability to connect with their context, and therefore to function well in it.

Contemporary software has many of these attributes. Socially impaired, repetitive, aloof, and inward, our software exhibits savant abilities and yet responds with tantrums or doesn't respond at all when taken out of familiar patterns. Software users "learn" the often-unreasonable requirements of systems and interfaces, which require that people make continual adaptations to avoid system tantrums. Like autistic children, most software is unaware of its context and therefore malfunctions when there is even a small mismatch between the assumptions built into the software and the realities of the environment.

The unique power of computing machines is that they are able to manipulate symbols mechanically, without regard to their meaning. Computers can handle repetitive tasks without making errors from getting fatigued, bored, or distracted. This power is also a severe limitation. Most software does not adapt to its environment, and mostly cannot. The inability of contemporary software to be context aware is, we believe, the crux of why software goals of dependability and reliability have been so hard to reach.

The very fact that we can name the symptoms and identify their sources also opens a path to finding a cure. We will cite examples of context-aware software that are free of these symptoms. They reveal the path toward post-autistic software.

3 Fitness

We mentioned in the introduction that the concern of the social dimension of software is the match between the software's actions and the practices, concerns, and expectations of the software's users. We will use the term *fitness* to evaluate in this dimension.

Since the earliest days of computer science, we have tied the success of programs to correctness. Mathematics seemed to create criteria of crystalline clarity of intended behavior and results of software. Engineering seemed to offer means to realize software objects in a complex world. But real world of users, with their idiosyncrasies, vagaries, and shifting interests, shatters the crystal. We have been stranded because we as a community have no rigorous tools and methods to address the social dimension.

Fitness, rather than correctness, ought to be the guiding principle for software in its social dimension. Fitness requires apprehending (getting the meaning of) the external environment of the software. Correctness looks at the internal consistency of software. Obviously, we do not want to eliminate correctness; but we do not want to apply it to the social dimension.

Every software system has an intended *domain of action* (DOA). (We will discuss the structure of domains of action later.) The software performs certain functions to assist users carry out actions within their range of expertise in the domain. Users rely on those functions to perform as advertised (correctness); and to refuse to perform under misuse or mistakes of use (fitness).

In its social dimension, modern software is autistic by the criteria medicine uses to diagnose this disease in people. The key fighting autism in software is to make software that can infer the user's context and adapt to it. This goal is

feasible: working examples of non-autistic software already exist. A careful analysis leads to the conclusion that these systems employ the principle of locality to infer the context of their use. The locality principle opens the possibility of significant progress toward fitness. Ultimately, software will store discoveries about its experience with users in knowledge structures and use it to adapt behavior to current context and reconfigure to future contexts.

Fred Brooks (2003) noted software at the human interface fails to be:

- Intuitive for the novice;
- Efficient in perception and motion for the expert;
- Robust under misuse;
- Facilitating in recovery from cognitive or manipulative mistakes;
- Helpful in diagnosing errors and suggesting corrective action; and
- Rich in incrementally learnable functions.

These are all important areas of misfit between software and the domains of action of its users. Context is important context for each of these. How can software recognize whether a novice or an expert uses it? What is misuse? What constitutes cognitive mistakes? What corrective actions make sense in the domain? What is the learning path of a practitioner in the domain? Brooks cites these as major challenges because he does not see in software engineering the intellectual framework that would allow a rigorous approach to them. The engineering tools and processes for software development have barely been able to provide more than token support for the social dimension. We will comment later on how the framework we will discuss can help with these challenges.

4 Case Study: Thrashing

It is instructive to examine one of the first cases of autistic software and how the autism was cured. This was the operating-system software that ran the first virtual memory systems in the early 1960s. That software was highly susceptible to *thrashing*, a catastrophic, unexpected collapse of system throughput. It was a major threat to the computing industry: who would buy a multimillion-dollar, high-performance computing system that could suddenly slow to an imperceptible crawl without apparent cause or provocation?

The story of thrashing and its defeat by the principle of locality has been well documented elsewhere (Denning 1980, 2006, 2006). Thrashing was a completely unexpected behavior. Engineers quickly determined that thrashing systems had fallen into a persistent state of constant paging, which they called "paging to death." But they had no idea what was causing it.

The solution came after Belady (1966) and Denning (1968) discovered the principle of locality and showed how operating systems could use it to measure the working sets of processes and guarantee them space in main memory. Operating systems incorporating working-set memory management did not, and could not, thrash.

In the terminology of this paper, thrashing was a form of "a system throwing a tantrum". By measuring working sets, the operating system could learn enough context to keep the peace among concurrent processes that would otherwise clash unproductively while trying to load their pages into a limited, shared memory.

Locality is the principle that executing processes tend to cluster their references into subsets of their objects for extended time intervals. It is a package of three interlocking concepts. (1) A program's dynamic behavior could be described as a sequence

$(L1,T1), (L2,T2), ..., (Li,Ti), ...$

of locality sets and holding times (also called phases). The locality sets are subsets of a *neighborhood*, which is the set of all objects required at one time or another by a process. (2) The locality sets consisted of all objects within a fixed "distance" from the computational observer. Distance can be temporal (e.g., time since prior reference to the object), spatial (e.g., the number of hops in a network to access the object), or cost (e.g., the storage cost of keeping the object in memory without using it). (3) Memory management is optimal when it guarantees each active program that its locality sets will be present in high-speed memory. The operating system maximizes throughput by caching locality sets close to the processor. Today, the ubiquity of caches stands as grand testimony to the demonstrated utility of the locality principle.

The locality principle has been adopted universally by hardware, operating systems, database, and network architects. It was rapidly adopted into practice, in ever widening circles:

- In virtual memory to organize caches for address translation and to design the replacement algorithms.
- In data caches for CPUs, originally as mainframes and now as microchips.
- In buffers between main memory and secondary memory devices.
- In buffers between computers and networks.
- In video boards to accelerate graphics displays.
- In modules that implement the information-hiding principle.
- In accounting and event logs in that monitor activities within a system.

- In alias lists that associate longer names or addresses with short nicknames.
- In the "most recently used" object lists of applications.
- In file systems, to organize indexes (e.g., B-trees) for fastest retrieval of file blocks.
- In database systems, to manage record-flows between levels of memory.
- In web browsers to hold recent web pages.
- In search engines to find the most relevant responses to queries.
- In classification systems that cluster related data elements into similarity classes.
- In spam filters, which infer which categories of email are in the user's locality space and which are not.
- In "spread spectrum" video streaming that bypasses network congestion and reduces the apparent distance to the video server.
- In "edge servers" to hold recent web pages accessed by anyone in an organization or geographical region.
- In the field of computer forensics to infer criminal motives and intent by correlating event records in many caches.
- In the field of network science by defining hierarchies of self-similar locality structures within complex power-law networks.

We will argue in the remainder of this essay that the locality principle is a powerful means of inferring context. The idea is that users function inside neighborhoods of objects, which can be inferred from event sequences of user actions. The inferred neighborhoods can be very useful to software that seeks to adapt to the user's context.

5 Autistic Software is Common

Most everyone can cite examples of software autism. In addition to thrashing, here are our favorites.

(1) Blue Screen of Death. The Windows operating system occasionally, without warning, completely freezes and the screen goes to a constant blue color. Only powering it off can restart the computer. All work since the most recent checkpoint is lost. Microsoft's experts believe the problem is caused by inconsistencies between the current version of the operating system and the thousands of device drivers supplied by third party vendors. Microsoft has mitigated (but not eliminated) the problem with a "certification program" that

checks device drivers for inconsistencies with the operating system and alerts the user. A similar, but much less common, occurrence is the Mac OS X "Kernel Panic".

(2) Voice recognition units. Large numbers of companies have turned to VRU's (voice recognition units). The vendors of these systems promise huge savings in call center staff -- because 85% of customer requests are typically routed to automated systems, the center needs only 15% of its pre-VRU levels. Many paths through these systems' dense voice-menu trees lead to automated systems consisting of a software agent interfacing with a database. Companies reason that many customers ultimately prefer automated systems in the same way they have come to prefer bank ATMs. But numerous surveys say that customers using these systems are not satisfied. They complain frequently of very long hold times, lack of options corresponding to their individual problems, surly and overworked service agents, and the blatant insincerity of the recordings provided ("we are experiencing unusual call volume, your call will be answered in the order received," and "we apologize for the delay, your business is very important to us"). Customers complain that the automated voice systems don't work well for anything but the most routine inquiries; the systems are easily confused if the customer does not use the right terminology, speaks too fast, or has a non-standard request. It is very difficult to get these systems to transfer to a live customer service agent. That so many users put up with these systems while despising them so intently is a massive example of adapting to autistic behavior.

(3) On-line help systems. Many businesses have turned to on-line systems for technical support. They no longer distribute printed manuals; instead they provide short help files behind a search engine that retrieves the files most relevant to keywords supplied by the user. Many have on-line, web-accessible "knowledge bases" containing thousands of documents and emails about troubleshooting various problems, again with a search engine front end. Many people find these systems marginally helpful at best. Customers who do not know the jargon often find that their keywords don't match the database. The searches produce dozens of matches that can take a long time to review and mostly do not answer the original question. They contain no way to search for situations that resemble the user's situation. Indeed, they do not even ask about the user's situation.

(4) Eliza and its descendants. In 1966, Joseph Weizenbaum created the program Eliza to carry on a conversation in the style of a Rogerian psychotherapist. It used simple context-free substitution rules to generate responses to keywords in user input. Although Weizenbaum intended it to discredit the Turing test because it was obviously unintelligent, many took it to be proof that Turing's conjecture of an intelligent system by the year 2000 was

feasible. Since 1991 the Loebner Prize has recognized the best Turing Test entrant. The winners fool their human interrogators for a few minutes at most and are not noticeably more intelligent than Eliza. The experience is of talking to a person who is easily distracted, zeroes in on words of no real importance to you in the conversation, and frequently changes the subject. These programs are hardly closer to the sustained-conversation goal than their predecessors 40 years ago. This has not stopped companies from installing similar, context-free voice recognition systems in their customer-service interfaces.

The common feature of these problems is lack of context awareness and the inability to gain relevant context information. Michael Dertouzos devoted his last book to these problems (Dertouzos 2002). He was concerned with the same kinds of behaviors that we have labeled autistic. He advocated a much more human-centered design process to alleviate those behaviors. For example, he advocated information devices that interact in natural language with users, data systems that record a user's information once and pass it to all programs that need it, tracking the trustworthiness of information sources, and personalizing software configurations. While such can be helpful, we believe they will not guarantee that the software will be context-aware.

6 Examples of Context Aware Software

Although much less common, context aware software does exist. The following examples are ordered by increasing sophistication of their context awareness.

(1) ATMs, Spreadsheets, Tax Preparers, etc. Some software is designed for the standard practices of a well-understood domain of action. The automatic teller machine (ATM) is a good example. It implements the standard actions of bank tellers -- deposits, withdrawals, account balances, transfers. The spreadsheet implements standard accounting practices -- storing numbers in columns, arithmetically manipulating numbers, calculating account balances, and the like. Tax preparation software implements standard tax preparer practices -- providing forms, interviewing taxpayers to find out what forms they need, transferring data to forms, and filing forms with tax authorities. Because the domain is explicit and well understood, these systems incorporate all the context information they need. They do not try to learn about the external environment.

(2) Mac OS X operating system. This system adapts to novices by offering a graphical interface based on a familiar metaphor of manipulating documents on a desktop. Advanced beginners can customize menus and shortcut keys and instantly find files containing any given keyword string.

Experts can access the Unix system at its kernel. The system does not detect a user's sill level; it accommodates advanced levels by offering functions that users can learn when they are ready.

(3) Thrashing controllers. Virtual memory systems measure working sets and guarantee executing processes enough space to contain their working sets, thereby optimizing system throughput and preventing thrashing. Shared communication channels can suffer from thrashing. When contention gets high enough that transmitters are likely to jam each other, the entire system can enter a persistent state in which transmitters cycle endlessly between trying, discovering a jam, and then retrying. Ethernet solved this problem with a back-off protocol that makes a transmitter wait progressively longer each time it retries unsuccessfully. A similar problem, with similar solution, was encountered in database systems when many concurrent transactions could contend for the same record lock.

(4) Linkers and Loaders. These workhorse systems have been part of operating systems environments since the 1960s. They gather library modules mentioned by a source program and link them together into a self-contained executable module. The libraries are neighborhoods of the source program.

(5) Bayesian spam filters. These filters use Bayes's law of conditional probability to guess whether a given email is in the user's locality space of interest or not. It builds its capability over time by watching which emails the user classifies as spam. The inference system learns from observations about the user rather than applying a pre-set filter.

(6) Semantic web. Semantic web, an R&D project of the World Wide Web Consortium (W3C), is a model of contexts that can be explicitly described. Using the extensible markup language (XML), one can explicitly declare structural relationships that constitute context of objects and their connections. Application programs can read and act on this information. An early example of this are that preferences in the Mac OS are stored as XML files so that the operating system and applications can instantly configure itself for the user.

(7) Google. This search engine gathers data from the Web. Web pages are ranked by a weighted combination of the ranks of other pages linked to them. Keyword queries retrieve lists of web pages containing the words ordered according to the ranking policy. In effect, this policy uses the links implanted by other users to infer a "community sense" of which web pages are most relevant given the keywords. It obviously works because most people say that Google gives them useful and relevant information very quickly and frequently on the first try.

(8) Amazon.com. This company was the first to offer a virtual world representing a first-class bookstore with over 2.5 million titles in stock. They pioneered the on-line shopping cart. They provided a database so that books could be looked up quickly with fragmentary information about author and title. They now fully content-index many books so that customers can find books containing phrases not mentioned in the title. They offer discussion groups and user reviews. The system collects data about user purchasing histories and recommends other purchases that resemble the user's previous purchases, or purchases by other, similar users. The Amazon.com virtual world reaches out and incorporates some user context.

(9) Games. Computer games produce imaginary worlds of action and draw players into them. The game remembers what the player has learned and adapts to the player's demonstrated level of skill. The game pushes users toward higher levels of skill. The same game can give different experiences to different players and can incorporate multiple players of different experiential backgrounds. Although games are best known for entertainment, they have been quite valuable as training aids. It is often the case that a user's world can be described as a game and thus a software game can be a valuable way to learn to play the real thing. Because games draw users into a world created by the game's author, game software creates and maintains its own context and does not have to make inferences about the user's context outside the gaming world.

(10) Forensics. At present, forensics is not a software system, but a human activity that infers criminal motives by correlating a lot of data across different observations of localities in which the criminal operated. This science has been a big success because of the ubiquity of caches in computer systems and networks. The caches were put there to optimize performance of applications. The totality of the event sequences recorded in them can be astonishingly revealing. Even when a user deletes a file, the storage medium, consistent with the principle of locality, still contains faint traces of the most recent versions; they can be retrieved by advanced signal processing methods.

The common feature of these examples is that the software is designed to fit its context. Except for the first case (ATMs etc), the software learns about its context of use and adapts to it. These systems collect data by observing the working sets of the software (thrashing), establishing direct connections to a user's neighborhood objects (linking), consulting user declarations in the environment (semantic web), observing a user's purchase patterns (Amazon), observing a user's skill at performing actions (games), and making inferences from cached event data (forensics).

Computer games are a special case. Games draw their users into a world created by the game's author. The game world embodies all the objects,

relationships, rules of action, and rules of strategy of the real social system simulated by the game. The game software thus defines its own context and does not have to make inferences about the user's external context. The other categories of context aware software are different because they make inferences about the user's world outside the software and adapt to it.

The vast majority of software is not this way. We believe this is so because that few people recognize the principles behind the successes.

7 Levels of Fitness

The examples above have been listed in an increasing order of their sophistication in gathering information about context and adapting to it. We have identified six distinct degrees of sophistication:

- Rank 0. Software is structurally designed to match a static analysis of the domain of action. No event traces are collected. Examples: ATM, spreadsheet, tax preparation, Mac OS.
- Rank 1. Event traces are recorded outside the application and acted on by outside agents. The application itself does not have to change or adapt. Examples: thrashing controllers.
- Rank 2. Event traces are recorded within the application, which passes the information to an outside agent for action. The application does not act on the context information it has gathered. Examples: linkers and loaders, spam filters.
- Rank 3. Event traces are recorded outside the application, then are read and acted on from within. The application relies on outside agents to learn context, and then acts on that information. Examples: Semantic Web, Google.
- Rank 4. Event traces are recorded within the application and are acted on from within. The application is self-contained with respect to adapting to its context. Example: Amazon.com, games.
- Rank 5. Event traces from multiple applications are integrated and correlated to form a larger picture of the intentions of the user of those applications. Example: forensics.

8 A Model of Locality for Apprehending Context

We said above that adaptation means to formulate actions based on analyses of relevant event traces. This is a sweeping statement. How do event traces reveal context? Do they reveal everything about context? What events must be

recorded? We do not want to leave the impression that we think that inferring context is easy. Although we cannot learn everything about context from event traces, what we can learn can be quite useful.

Inferring context is an inherently hard problem. The philosopher Martin Heidegger devoted an entire book to the subject, concluding that action always occurs in a framework of interpretation, that the framework depends on the observer's history, that the rules and assumptions of the framework can never be completely revealed, and that a person's actions can reveal some context of which the user is unaware (Dreyfus 1990). Noting that computing machines are capable only of processing context-free rules, Winograd and Flores showed software will fail if is expected to grasp context (1987). Their main example was expert systems, which try to simulate behavior of human experts; but lacking the ability to sense and interpret context, the system's performance cannot match the human expert's. Dreyfus cites the example of the Cyc system, an effort to accumulate trillions of facts about the world in the hope that a logic system with access to all those facts would exhibit common sense (Dreyfus 2001). He does not think this will happen: common sense is not reducible to facts.

All this leads to unnecessarily pessimistic conclusions about the ability of software to learn enough about human context to be useful. From long experience with the principle of locality, however, we think such conclusions are unjustified.

In its original form, locality meant that computations clustered their references into subsets of their pages. The working set inferred the contents of locality sets by recording page references in a backward window. Over time, we extended the principle into many areas. In all cases the systems infer a neighborhood of objects based on observations of what objects the computation was actually using. The inferred neighborhoods are stored in caches for fast access by the computation.

In the case of humans interacting with software, the neighborhoods belong to the user, not the software; and they depend on the user's level of skill. Thus our model of context of human interaction with software is (1) A domain of action in which (2) an observer operates (3) at an expertise level (4) within a set of neighborhoods and (5) with an expectation of optimal performance if neighborhood objects are cached nearby. These five components are elaborated below. (See Fig. 1.)

(1) The domain of action is the scope in which players carry out moves in pursuit of some overall purpose.

(2) The observer is the user who is trying to accomplish tasks with the help of software, and who places expectations on its function and performance. In some cases, especially when a program is designed to compute a precise, mathematical function that is the same in all contexts, the observer is built into the software itself.

(3) A level of expertise is a degree of skill that a user demonstrates in the domain of action. The criteria for these levels are part of the domain definition. In general terms, the three most common levels are:

- A *novice* sees only the rules of acceptable or disallowed moves in the domain; action consists in the applications of rules.
- A *competent* user sees situations and applies associated rule sets, mostly without thinking; action consists in satisfying requests from other people in the domain.
- An *expert* is able to expand the repertoire of situations in which he can act proficiently, is able to alter the rules applying to a situation, and is able to devise new strategies based on overall assessments of action flow in the domain.

User expectations and modes of interaction will vary according to their skill levels. Some software is already designed to accommodate a range of user skill levels, as we noted earlier in our example of the Mac OS X operating system.

(4) A neighborhood is a relation linking an observer to objects; some objects may be in the same computer system as the observer, others distributed throughout the Internet. An observer operates in one or more neighborhoods while using the software to carry out or coordinate actions. The language used by the observer to express actions depends on the neighborhood; the actions that cause state changes in the neighborhood can be observed and recorded as event records in the observer's computer system.

(5) The expectation of optimality is that the software will complete work in the shortest time if neighborhood objects are ready accessible in nearby caches. For practical software it is impossible to know the full contents of neighborhoods. This does not matter because the user is likely to need only a small subset of a neighborhood at a time. The principle of locality comes into play for inferring the subsets of neighborhoods that must be cached. The inference will correlate the (dynamic) event trace with other (static) information available such as values of environment variables, declarations about structures of objects, current time and place, and criteria for levels of expertise.

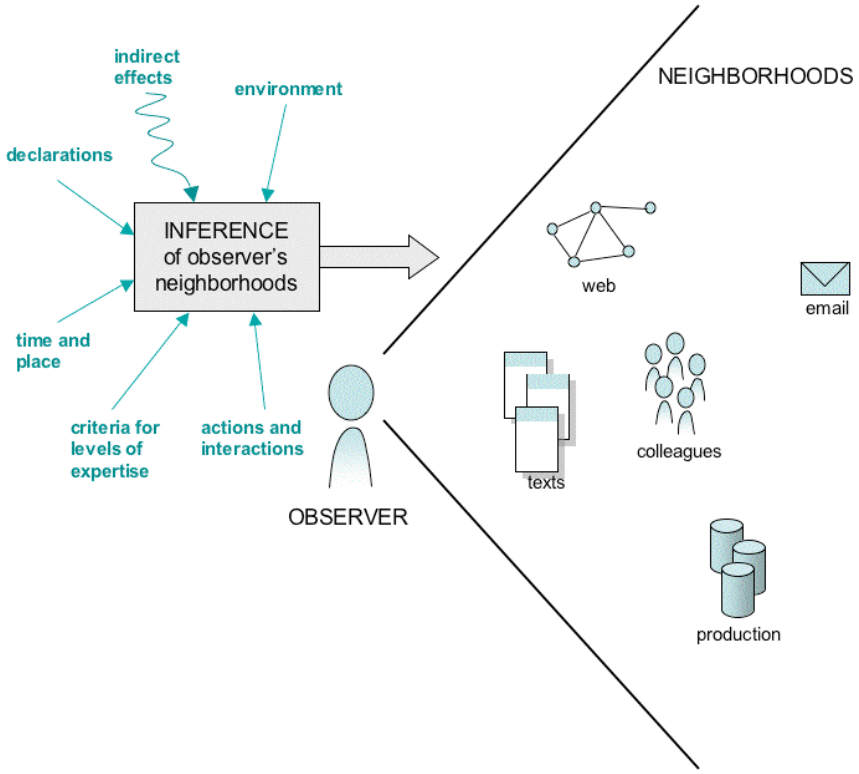

Fig. 1. The modern view of locality is a means of inferring the context of an observer using software, so that the software can dynamically adapt its actions to produce optimal behavior for the observer.

9 Designing for Fitness

The critical step toward context-aware (Post-Autistic) software involves discovery of the observer's world outside the software and maintaining an awareness of it. Virtual memory, one of the earliest developments in computer systems, illustrates how this might work and suggests a path that could work for other software. First-generation virtual memory systems exhibited thrashing because the virtual memory software had no insight into the actual amount of memory that application software needed for efficient operation. With the locality principle, second-generation virtual memory software could estimate the neighborhoods by monitoring recent past page references. By caching the estimated neighborhoods close to the processor, this scheme also yielded near-optimal throughput.

The locality principle worked because processes had significant holding times in neighborhoods. The recent past references revealed the content of the neighborhood; the long holding times made it quite likely that in the immediate future the program would access the same neighborhood. The same is true for a user's neighborhoods. We can record event traces (from inside or outside the application) and use the recordings to estimate the user's current neighborhoods. The application software or its runtime system can usefully act on that information because the neighborhoods do not change too fast.

Today's systems yield richer sources of reference events than early storage systems. They can monitor accesses to database references, files, devices, and web objects. They can also correlate event sequences of different, but interacting observers, to learn about common neighborhoods.

When the software examines event-trace data to estimate neighborhoods, it can apply a distance metric to decide which objects mentioned in the trace are close to the observer. As in storage systems, the distance metric can be temporal, spatial, or cost, whatever is most convenient or effective for the type of neighborhood.

However, there is no reason to limit neighborhood estimation to simple distance measures. Any kind of inference can be useful. Bayesian spam filters illustrate that Bayesian inference can estimate the most likely hidden states (spam or not) given the past data about which objects the observer considers to be spam.

Just knowing the neighborhoods in which an observer is operating can enable the software to adapt and exhibit enough context-awareness to be useful and not autistic. It can also help optimize performance because the objects in current neighborhoods can be placed into caches for the fastest possible access.

Ultimately, the road to Post-Autistic Software will go past immediate discovery-response behavior. Advanced social responses require that software store its discovered knowledge in structures that can be referenced in future situations. If the new situation resembles a past one, the software can propose a similar action. The software can adapt and reconfigure the structure as more data are collected. If a user's current actions do not fit the expected profile, the software could query the user -- for example, "Never before have you sent email messages to every one in your address book – are 'you' there?" This can work in the opposite direction as well: if software is not responding, the user could query the accompanying social structure to get ideas on what is wrong.

Some readers will be concerned that context-aware software may pose certain privacy risks. Given the current dismal record of organizations at safeguarding personal information and of issuing software that surreptitiously

collects it, distrusting them is understandable. Some people would go further and say that it would be prudent to keep our software unintelligent, in which case we would need only to confront software stupidity rather than software autism.

These are really the same risks and concerns that arise around all software systems. We believe that context-aware software as described here may actually help *decrease* the risks because it makes explicit that the software is measuring context and forces the maker to adhere to higher standards for privacy protection.

We believe that, as with any software function, context-awareness ought not be part of software unless there is value in having it. We are aware that autistic software is common and causes much grief and frustration. It seems that many people actually want software to be more aware of context and more intelligent, less likely to cause breakdowns for its users. The consumer demand is already building and software makers will move toward more context-aware software. We hope that they learn to do so intelligently and in a way that mitigates risks.

10 Concluding Remarks

We have proposed that a new criterion, fitness, augment the historical criterion of correctness in our thinking about well-functioning software. Fitness measures how well the software is able to adjust to the intentions and expectations of its users -- that is, to the context of its use. We proposed the principle of locality as a tool for achieving fitness because it gives the means to infer the neighborhoods in which a user is operating. Much software today is a poor fit with user situations and intentions. The existing examples of context-aware software suggest that locality principles can help software gather event data, infer user neighborhoods, and adapt its structure and behavior to align with the user. Designing for fitness would close the gap between software's *presumptions* (built in by designers) and the user's *intent*. Table 2 summarizes how the software designer can learn about the elements of context used by the application.

At the beginning of the article we mentioned six challenges that Fred Brooks saw for those designing interactive software. Context aware software as discussed here can go a long way toward meeting Brooks' challenges. See Table 3.

We offer these observations and speculations, not as a final answer to the challenge of context-aware software, but as a promising direction for exploration and research.

Table 2: Discovering the elements of context

Item	**Means of discovery**
Domain of action	Prior (static) analysis of domain
Criteria for levels of expertise	Prior (static) analysis of domain
User's level of expertise	(1) watch actions, compare with criteria (2) user declares (In either case, system adapts by hiding or revealing functions.)
Neighborhoods	(1) tagging objects by class (static) (2) inference from event traces (locality)

Table 3: Context-Aware Software and the Design Challenges

Brooks' Challenges for Discipline of Design	**How Context-Aware Software Meets the Challenges**
Intuitive for novice to learn	Software detects presence of novice and guides choices offered
Efficient for expert to use	Software detects expert presence and clears out interface noise
Robust against misuse	Software compares requested actions for fit to domain of action, warns of anomalies
Helpful in resolving errors	Software calls attention to atypical or ruinous steps
Incrementally learnable functions	User interface presents choices, action paths, and functions consistent with user's current level of expertise
Facilitates recover from manipulative or cognitive mistakes	Software assess requested actions for fit, warns of potential ruinous ones; can be trained to learn common mistake patterns and corresponding recovery patterns

11 Bibliography

Belady, L. A. "A study of replacement algorithms for virtual storage computers. *IBM Systems J. 5*, 2 (1966), 78-101.

Brooks, F. P. Jr. Three great challenges for half-century old computer science. *J. ACM 50*, 1 (Jan 2003), 25-26.

Denning, P. J. The working set model for program behavior. *ACM Communications 11*, 5 (May 1968), 323-333.

Denning, P. J. Working sets past and present. *IEEE Transactions on Software Engineering SE-6*, 1 (January 1980), 64-84.

Denning, P. The Principle of Locality. *ACM Communications 48*, 7 (July 2005), 19-24.

Denning, P. Locality. In *Festschrift Honoring Erol Gelenbe* (J. Barria, ed.). Imperial College Press (2006). To appear.

Dertouzos, M. *The Unfinished Revolution.* Harper Business (2002).

Dreyfus, H. *Being in the World: A commentary on Heidegger's Being and Time.* MIT Press (1990).

Dreyfus, H. *On the Internet.* Routledge (2001).

Winograd, T., and F. Flores. *Understanding Computers and Cognition.* Addison-Wesley (1987).