# Portable REXX Applications and Reusable Design

Edmond Pruul

"Portable Rexx Applications and Reusable Design"

Edmond A. Pruul

RD 1 Box 632
Afton NY 13730
USA
Electronic Mail: p00146@psilink.com
Voice Mail: 1-607-693-1030

ABSTRACT

The application owner and developer want to port their applications to new operating system environments. The Rexx language offers inherent advantages: readability, an active Standards organization, available source code, good input-parsing functions, easy source-level debugging; but no practical breakthrus have been identified in the Rexx community. The same problems that plague application portability in general apply to Rexx programs also: unwieldy code for several operating system environments; ownership conundrums; interface confoundment; and many other problems, old and new. Using Rexx as an example of a language whose applications should port easily, reveals the intractability of the portability problem. Reusable Code is problematic in startlingly similar ways. Reusable Design is a promising paradigm for a general attack on the Reusable Code problem. A Rexx application, being readable (accessible) by the average programmer is a possible stage to experiment, in a practical way, with Reusable Design. The Reusable Design paradigm is based on the classic principles of modularity in Computer Science. It can include object based or object oriented methods but the prime principle is semantic as well as syntactic readability -- the actions of the Reusable function are clear and concise to programmer. Readability allows early planning by potential reusers -- customers for a reusable function. As the Rexx Application developers rely more on reusable components, market forces could encourage the proliferation of popular reusable components to popular operating systems.

## PREFACE

The thoughts and experience herein are those of an operating system coder and designer from 1968 to 1992 and do not pretend to be current in this year 1994.

## PORTABLE Rexx APPLICATIONS AND REUSABLE DESIGN

Outline.

1. Porting Applications: motivation.
2. Measuring Success: when to stop?
3. Code Reuse: promise and problems
4. It is the Design, stupid.
5. Why Rexx?

## PORTING APPLICATIONS: WHY BOTHER?

Definitions and 'ground rules' help address a problem. '**To port**' means to change a product such that it works in two or more environments. An example of a portable product is a 'Walkman' -- a personal tape player. A Walkman ports very easily around the world; the tape player's motor runs on DC batteries so the local power system's voltage and frequency are irrelevant to the Walkman. An '**application**' is a complete -- not a partial -- product. We focus on customer-related computer products. A complete operating system is an application. Rexx is an application. A PS/2 is not since it is bare metal -- it is not complete. A pre-loaded PS/2 is an application. Note the definition is broad. A ground rule in problem solving is to ask whose eyes to use. One could say we must know the scope of the problem or perhaps its environment. Our scope is strictly a business viewpoint -- a marketplace. We shall examine moving computer related products to another marketplace.

Compare your notions of 'Why Bother?' with these.

_ Increase Marketplace Share. One would think making that increased profits result from increasing market-share. What metric will predict our success? Will the cost of development and maintenance exceed revenues? That is the question.

_ Promote brand name recognition: "We have it all!" Or Foot-in-the-Door Syndrome: "Some day we will have it all on your computer."

_ Protect the product owner's other products in target. Spread development costs when one market would not profit the product owner. When American engineers looked at designing a small personal tape player they may have thought.

"We can not make a profit in the '60 hertz, 110 volt' marketplace. The cost of a port would be too high. Light weight batteries would not be powerful enough. We must package a different motor. We have no idea how to write diagnostic messages in Kanji or Cyrillic. (Add your own problems here)."

_ Clone a nifty application for my computer! Is cloning market related? Maybe. Examine the cloning of applications. What are our real motives? Are we violating patents or depriving someone of their copyrights?

## MEASURING SUCCESS

Quality is the obvious metric and we know that surveys measure quality reasonably well but customer surveys are not predictive. What exactly does quality mean? Does quality mean delivering on time; or delivering a product that works as well as we can make it; or delivering what the customer wants? They all are good goals that any product should meet if possible. Assuming we could measure our product using these three quality goals how do we weigh the three against each other. We know that one of the definitions eventually must take priority. The developer can not decide if a port will be successful until we know which definition of quality the target marketplace demands.

Consider these metrics vis-a-vis porting applications.

_ Product makes a profit. Sadly profit is not a timely metric.
_ Maintenance cost: we can predict simple costs such as help lines, change teams, continuous market research, advertising, code control systems, legal fees. Often measuring failure is possible while measuring success eludes us.
_ The product looks and feels the same in both environments.
_ The product looks and feels like other applications in the new environment.

## CODE REUSE: WHY IS IT PROBLEMATIC?

Does the Walkman have reusable components? What happens when the batteries run down? The engineers decide to put a DC plug in the Walkman so that anyone can buy a Reusable Component called an AC/DC Converter. The problem is there is no common voltage for battery powered appliances. Every engineer picked a different voltage -- 9.65, 13.1 and other peculiar voltages. Good try, engineers. Perhaps a variable voltage converter would be a better Reusable Component.

Think of examples of good reusable code: string.h in classic C libraries, Rexx functions such as STRIP or WATCOM's VXRexx, a 'visual editor' for Rexx on OS/2. Intuitively reusable components will be clearly useful if there is a big gain. The **mass** attribute could be due to many potential reusers or big functions replacing large amounts of new code. Consider the Rexx interpreter. It is an excellent example of a reusable 'scripting' component. Why? The Rexx reuser gets much more than originally specified, or **serendipity**. Rexx is massive since it replaces large amounts of code, Rexx is **mature**; Rexx is used by millions[1] of amateur programmers Rexx is **robust**; it does not break. Another good example is IBM's XEDIT used as an application base; the reuser's customers gain strong editing and searching function gratis.

The most gain for Reuse components is from serendipity and mass. Maturity and robustness are problematic.

Good designers know how to design things when they are expert. In practice the problems of general-purpose code-reuse by an average programmer are overwhelming. Consider these problems.

_ Maintenance: who maintains components; how to compensate the maintainer; how to control many versions[2]; lose of intellectual control as time passes and persons pass on to new jobs and the next life; delivery of new function and service including preventive service.

_ Disappearing customer base: First, our reuse candidate loses a prospective customer. So development stops. Next month a new customer surfaces, the project restarts only to disappear again.

_ How to measure reliability or quality of a reusable code component. How does customer convince management to trust reusable code? What would be the service cost projection?

_ Publicity: how, where, and issues of truth in advertising touch on personal sensitivities.

_ Packaging: When would we bind reusable components to the reuser's program? It could bind when compiled, at product build, at installation of the product, when the application loads or at run-time.

_ Myopic design and semantic provincialism: a coworker needed a subroutine to test, in a secure way, if a person is a "SFS Administrator." The words mislead. In fact, the programmer wrote a routine to test if a process-id is acceptable by a named resource manager for a certain specific authority. The word "SFS" is superfluous. The word "administrator" implies a permanent attribution of a human being. Worse, this label implies the reusable component has some power to enforce or guarantee its response for some un-stated period of time. False, the answer is advisory only. The power of authorization remains with the resource manager's authorization mechanism.

## REUSABLE DESIGN: CAN WE HAVE SERENDIPITY AND MASS?

Reusable design could mean "good external design." Good syntax is a given: simple, targeted for performance,[3] no surprises and no side effects. Semantic clarity is the rub. Cultural tunnel vision is problematic by definition. The cure is an accessible and readable design. Early disclosure and serious attention to criticism are good; continuous disclosure is better. Rexx Library functions are outstanding example of reusable components; the required attributes are present: one responsible person, expert in the field, serious helpful customers.

What can we do?

_ Study and understand today's and tomorrow's methodologies: Temporal Logic; Gries' Axiomatic technique; SMALLTALK; Finite State Machines; Data Flow Analysis; and Event Analysis amongst many others.
_ Buy and read books.
_ Take all the design courses available; retake them a few years later.
_ Practice off the job.
_ Volunteer for inspections. The more design and code we study the wiser we are.
_ Join the local Reuse Advisory Board, evaluate reuse code candidates.
_ Read code and designs.
_ Join or get advice from the local Wisemen Council.

WHY Rexx?

_ Standards Group is in-place and active. Rexx semantics as well as syntax are consistent.

_ Slivers are easy to implement.

A sliver is the slimmest possible layer between a portable application and the complete computer system it would run on. Syntax errors will occur in the sliver. Semantic errors are harder find and harder to isolate. "The final" semantic error may be impossible to find; An application can not port to environments that are semantically incomprehensible to the original environment.

_ Universality: Rexx will run on all new operating systems[4].

_ Readability or accessibility: Rexx is justly famous for first run successes. Problems in Rexx code are rare. Errors are most likely when calling system commands. The author's first Rexx program worked perfectly on its first test; in Rexx circles this experience is not a surprise.

_ Debuggability: Rexx has implicit symbolic debugging; Rexx programs can be distributed as human readable code. There is an optional compiler but normally Rexx programs are interpreted from the source code. Anyone could read the program to solve a problem and test a fix.

---

1 This number seems too large but the author extrapolated it from an estimate by Bill Fischofer in 1991. He calculated the number of VM users to be 30 million.

2 Need strong code control systems.

3 A Rexx example is LEFT and SUBSTR, the former being a special case of the latter.

4 Author's opinion.