The Theory and Practice of Adaptive Components

Paul G. Bassett
pbassett@netron.com
October 11, 2000
Erfurt, Germany

(http://www.netron.com)

Overview

- Origins of adaptive components
- How do they work?
- Why do they work?
- How well do they work?
- What does the future hold?

Origins

- Subroutines, Macros, & Generators
  - all have been around since the early days of computing
- Curiously, macros declined as subroutines ascended
- Let's try to understand why
  - compare and contrast macros with subroutines
  - we'll come back to generators later



Objections to Macros

- Global Variable Problem?
  - macros overwrite each other's variables (as did subroutines)
  - 3GLs acquired block structure
    - "local overrides global" scope
  - Macros can be scoped too
- Small Memory Problem?
  - true 20 years ago; false today
- Macros are Superfluous?
  - subroutines emit data
  - macros emit program structures
    - including subroutines!!
  - macros are *meta-components*

Build-Run Duality

- Each run-time aspect has its construction-time dual
- Duality is the source of simplification and insight
  - e.g. one macro *functional* can replace an infinity of (polymorphic) *functions*
  - e.g. each feature of a run-time architecture has its construction-time architecture dual:
    - encapsulation -- change isolation
    - functionality -- generality
    - ease-of-use -- adaptability
    - efficiency -- complexity

Engineering Software for Softness

- Software's Raison-d'être: Ease of Modification
- This key property remains largely informal. Why?
  - We pay a huge price in so-called "maintenance"
- Humans notoriously poor at fussy software changes
  - non-linear: small changes can have large effects
  - "brittle software"
- Conventional wisdom is to treat components as immutable -- Lego Blocks

Ill-defined Problem Domains

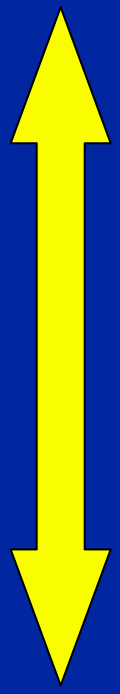
- Domains range from well-defined to …
- Ill-defined:
  - properties are not well understood
  - conflicting constraints
  - unstable with time
- Ill-defined domains eventually become well-defined
- Even well defined domains benefit from adaptability

Why have we done such a poor job of engineering software for modifiability?

- Eons of evolution
  - blinds us to non-physical degrees of freedom
- Psychological need
  - to believe only we can control non-linearity
- A lack of appreciation for context-sensitivity
  - just because we can build software from fixed parts doesn't mean we should

Context:

Sensitive

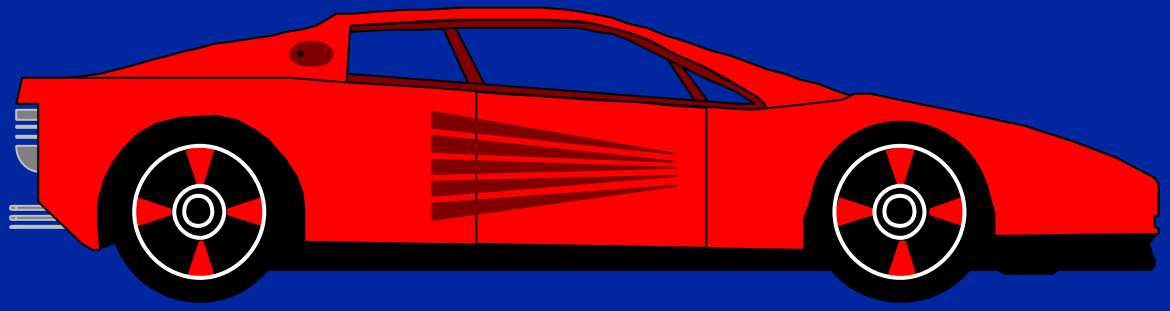


Free

- Context-free parts:
  - make a few, simple assumptions re surroundings
  - can satisfy more contexts as-is
  - desirable to a 1st approximation
- Problem: mutual ignorance leads to clashing parts
  - not the fault of any one part!
- "Solutions" focus on parts
  - modify parts → retrofit hassles
  - subclassing → sea of look-alikes
  - work-around → inefficiency
  - all "solutions" boost complexity

Adaptive Components: An Analogy

- Imagine assembling any make or model of car by adapting one generic fender, one bumper, one engine, etc.
  - a silly idea in the physical world
  - makes perfect sense in software assemblies



Adaptive Component: A Definition

- A formal embodiment of an *archetype* together with *rules* for combining (sub)component-archetypes into context-specific structures and / or functions
  - Applies to any domain whose elements can be described by a symbolic language
  - Generic adaptive components are roots of nouns and verbs in a domain-specific vocabulary

From Archetype to Adaptive Component

- "the original pattern or model from which all other things of the same kind are made"
- To create a formal archetype:
  - partition domain into (sub) categories of similar nouns and verbs
  - for each category devise one typical model / solution, or more
  - parameterize it such that each element is a default value
  - decompose a model if it contains reusable sub-models
- Adaptive components are "forgiving": You don't have to "get it right the first time."

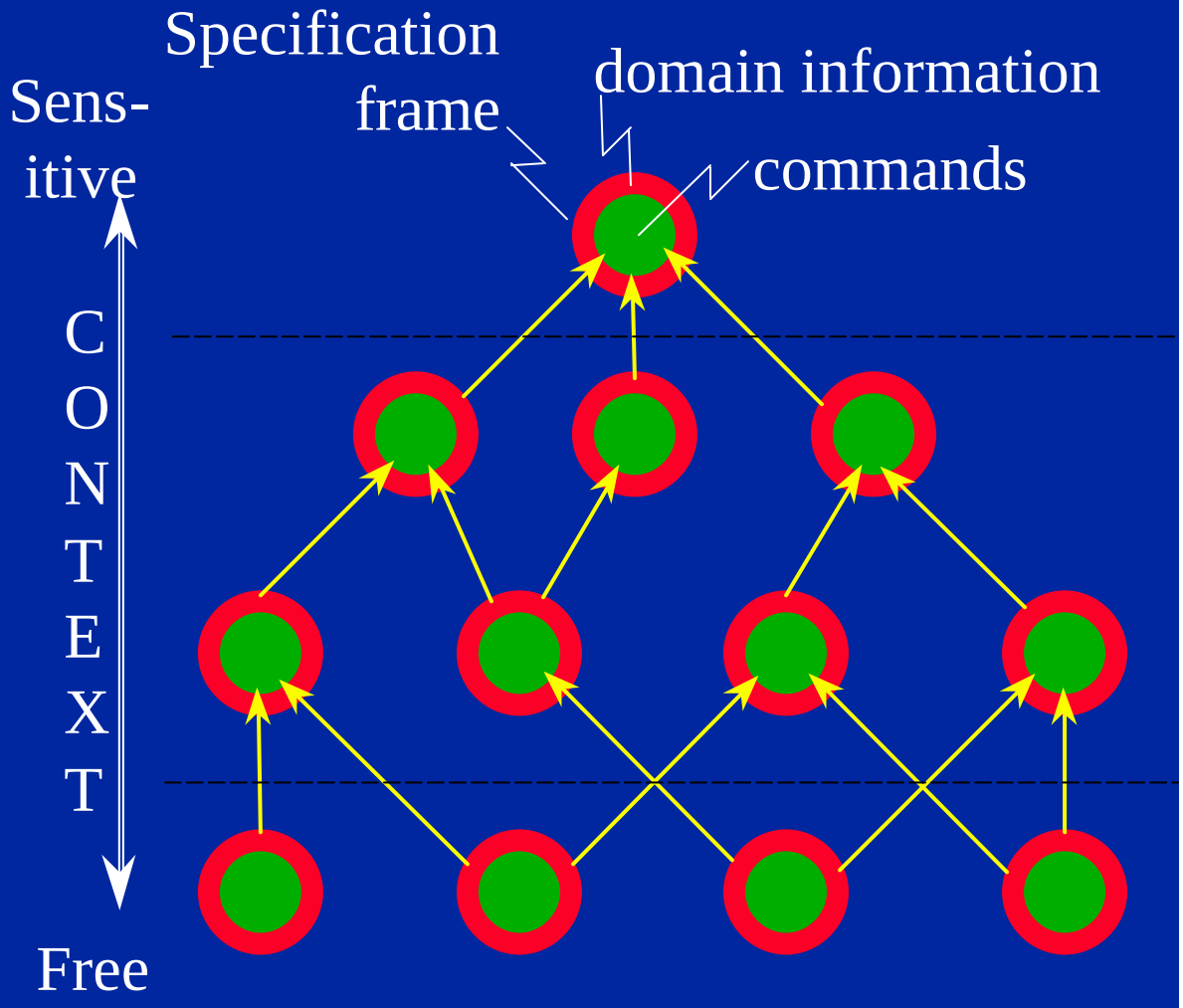
Raisons d'être of Adaptive Components

- Coping with unpredictable change
  - avoid premature decisions re what is common and what is variable
  - alter components after reusing them, without side-effects
- Automating modifications
  - more safely than by hand
  - much faster
  - construction becomes an analysis tool
  - maintenance disappears as a distinct process

A Technical Implementation

- ACs are special macros, marking up some underlying language
  - XML could be used
  - Frame technology is used
- Frames form *semilattices*
  - processed as if they were trees, starting at a root
- Frame processor (FP)
  - reads frame (sub) assemblies
  - interprets mark-ups
  - emits fully customized, base language texts

Assembling a Module From a set of Nested Contexts



- A frame can select, add, modify, delete, iterate, and instantiate *any* of its sub-assembly frames.
- A frame's parameter settings take effect if and only if it is the first of the currently active frames to set them.

Frame Structure

<frame> **name** <body> **frame-text** </body> </frame>

- Frame-text = mark-up + plain
- Plain text = literals = not yet known to vary
  - emitted as is

Handling Variability: Parameter Assignments

<replace> name <by> expression </by> </replace>

- name can be an expression that evaluates to a name
  - indirect references, indexing
- rhs expression may be ignored!

Scoping: Global overrides Local

- If in frame F parameter P is undefined, only F can <replace> P
  - when F exits all parameters it <replace>d are undefined
- "Same as, except" solutions
  - parameter settings are minimized
- Frames are unaware of their "reuse contexts"
  - allows for "incremental variability"
- The higher in the hierarchy, the more clout a frame has!

Marking up text with <replace> parameters

■ <;> **name** < / ;>

- name can be an expression
  - e.g., a composite name

•Makes data structures, interface definitions, and logic context-sensitive

•Also controls the processing of frames

Adapting Subassemblies

<adapt **[**samelevel**]>** **frame-name** [<insert [before, after]> **break-name**
<content> **frame-text** </content>
</insert> **…]**
</adapt>

<break> **name** <default> **frame-text** </default> </break>

- <insert>s match to <break>s anywhere in tree
  - evaluation occurs in the local context
  - frame engineer can think of an entire subassembly as a single frame
- samelevel enables reusable parameter settings

The World is Arbitrarily Hairy!

- Why have <insert>s?
  - must be able to cope with unpredictable changes
  - every element in a frame can be a default <break>
    - “incremental variability”
    - dangerous, but less so than what programmers do

Select is a Case Statement

<select> <;>**name**</;>
<when> **relational-expression*** <content> **frame-text** </content> …
[<otherwise> **frame-text** </otherwise>]
</select>

*one of: < = > ≤ ≠ ≥

- Used for:
  - hiding known variations at *a single point* in the tree
  - localizing a variation that impacts *multiple points* in the tree
    - e.g., targeting multiple platforms, language dialects
  - evolving components *invisibly to existing* systems
    - avoids polymorphic proliferation, work-arounds, inefficiency

Archetype Iteration

<while> <;>**name**</;> <content> **frame-text** </content></while>

- Used for handling a large or unknown number of patterned repeats
  - e.g., operator entry of data fields
- Terminates when <;>name</;> undefined

Generators are Harmful and Unnecessary!

- Generators:
  - presume to know á priori what's needed in all contexts
    - chronic lament -- either changes are verboten or they get destroyed
  - they *internalize* domain semantics
- Adaptive components *externalize* semantics
- Generators become bi-directional translators that emit parameters to externalized semantics
  - SEs are free to customize any details
  - Users are free to regenerate without loss of customizations

Adaptive Components in Practice

- What does AC theory offer?
  - Meta-models for domains from well- to ill-defined
  - Automated software modification with increased reliability
  - The ability to collapse the information needed to span a given domain
  - Modularity that matches the domain
  - Flexible software architectures (product lines)
  - Adaptability across time
  - Localization of context-sensitive domain knowledge
  - More forgiving approaches to analysis and design
  - The means to transcend code generators
- Net result:
  - less time and effort to develop / evolve systems

Study Background

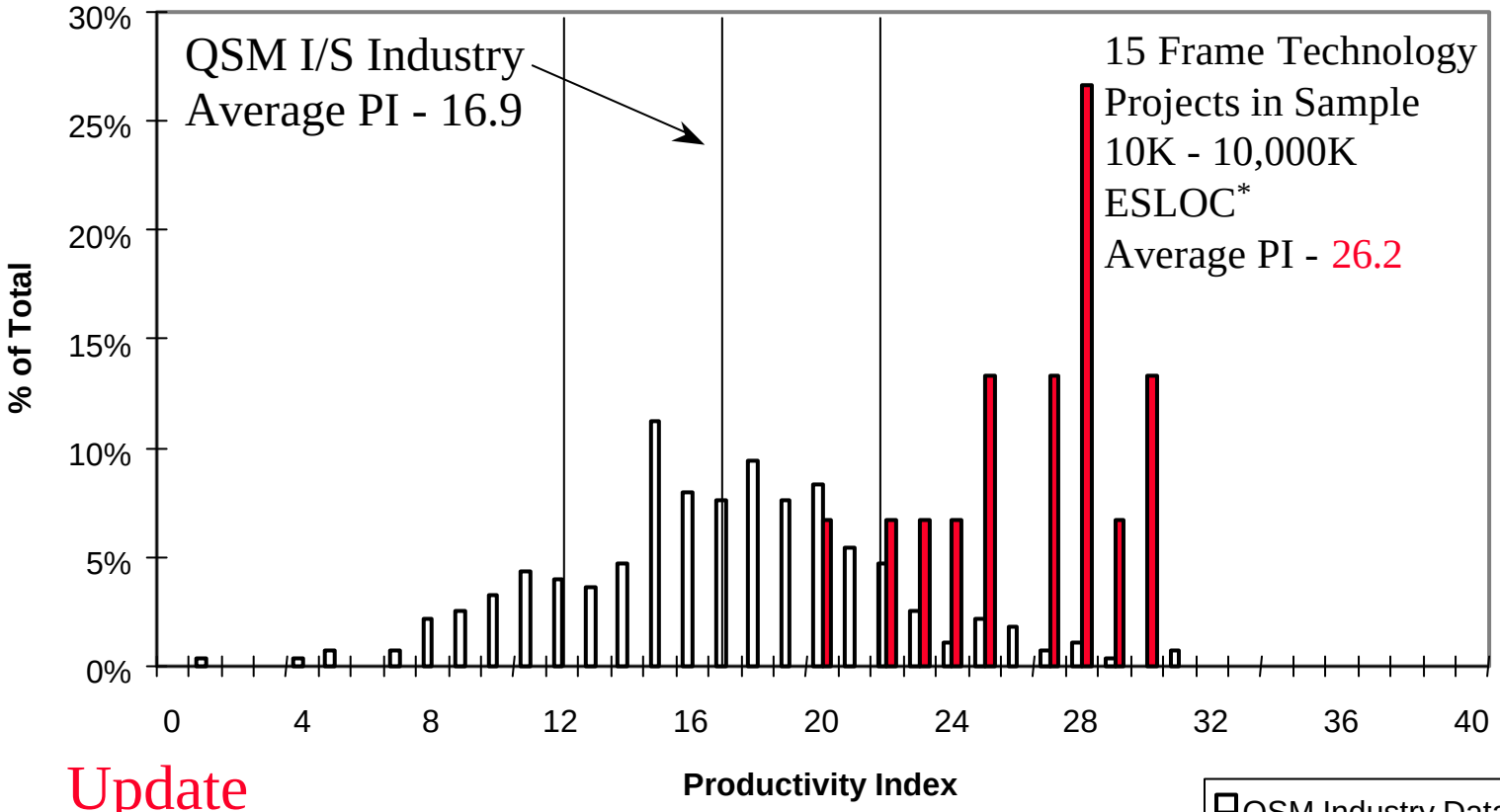


9 Organizations (in 7 Industries) Funded and Participated in the Study (2 Wish to Remain Anonymous)

- Ameritech
- Chemical Bank
- Hudson's Bay Company
- Noma - Cable Tech
- Revenue Canada
- Teleglobe Insurance Systems
- Union Gas

Frame Technology PI Vs. Industry

Productivity Index Comparison
Netron Projects Vs. QSM I/S Industry Average



Update

As of Jan 1 97: 29 project average PI = 26.9

*ESLOC = effective source lines of code without blanks or comments

Copyright QSM, Inc. All Rights Reserved



QSM's Productivity Index

ESLOC = effective source lines of code (no blanks / comments)

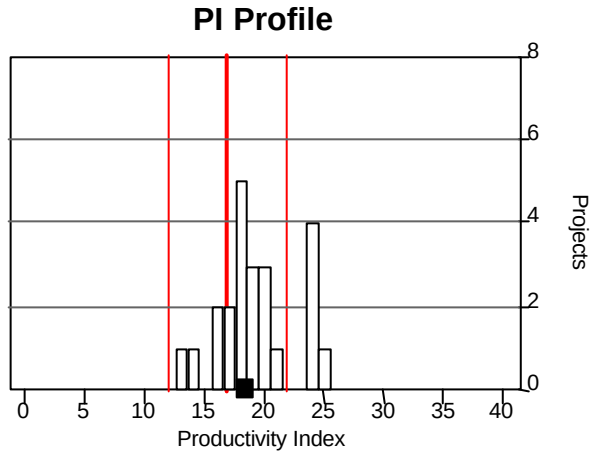
time = detailed design to user acceptance, incl.

effort = staff months

B = ESLOC related skills factor (0.16 - 0.39)

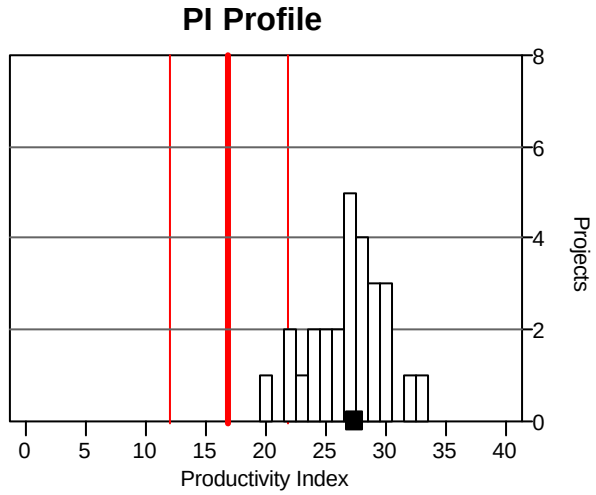
$$PI = \log_{1.272}\left[\frac{ESLOC}{time^{4/3} * (effort / B)^{1/3}}\right] - 26.6$$

OO PI Comparison: Class-based To Frame-based



Class-based Project Sample

■ = sample average = 19.2



Frame-based Project Sample

■ = sample average = 26.9

If adaptive components are so good, why are they not standard practice??

- What does it take?
  - Sensitivity to the need to model context-sensitivity
  - Recognition that machine-assisted code changes are much more reliable
  - A focus on avoiding the “brittle software” syndrome
  - ...

If you agree, then help make it happen!

Thank You!

Questions Please!