P2: A Twenty+ Year Retrospective

Don Batory
Department of Computer Science
University of Texas at Austin

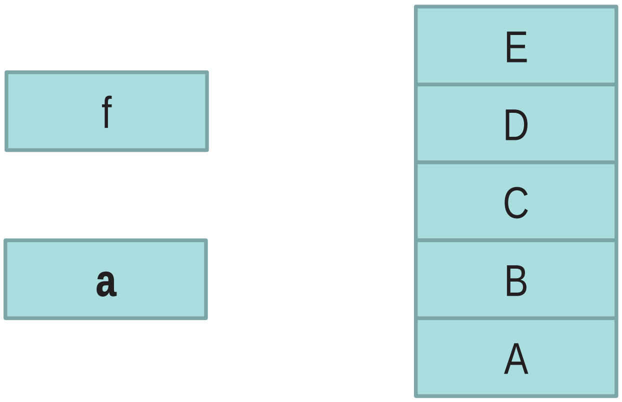
April 2012

Introduction

- For 30 years, I explored software architectures, component-based systems, software product lines, DSLs, and automated program development
- My background was Relational Database Systems and how DBMSs were built
- In the 1990s, I transitioned to "Software Engineering"
- My contribution is to explain that programs are complex structures and whose construction follows simple and elegant ideas in mathematics
    - I take a broad, unified view of contributions in software design over the last 40 years – there is a reason why engineers and designers do what they do – it is not by accident, but part of a larger scheme where elementary mathematics plays a pivotal role in automated program development

Central Observation

- Complex software has levels of abstraction – layering – which when designed properly leads to a compositional model of program constuction
- Focusses on a fundamental approach to the modularization of increments in functionality (features, collaborations, transformations)
- Construct customized systems hierarchically, evolve though exchanges



Ideas Scale…

• 1986 database systems	80K LOC
• 1989 network protocols	
• 1993 data structures	**P2**
• 1994 avionics	
• 1997 extensible Java preprocessors	40K LOC
• 1998 radio ergonomics	
• 2000 program verification tools	
• 2002 fire support simulators	
• 2003 AHEAD tool suite	250K LOC
• 2004 robotics controllers	
• 2006 web portlets	
• …	

P2

- Is one of a series of tools (P1, P2, P3, DiSTiL) that generated data structures
- Work circa 1991~1998

- Motivation: **Booch components** (prior to STL) were available, and I thought were awful. Little or no ability to compose data structures, little or no ability to swap data structures, low-level programming, not declarative, and essentially, all the hard work is still a burden on programmers

    - 20 years later, not much has changed

- Relational database technology provided a very powerful solution to some of the more common and complex programming tasks – P2 was our take on it

Basic Ideas of P2

- Straight from RDBMS: raise the level of programming to querying and updating tables. Tables are a clean programming abstraction with iterators to return results of a declarative query

- Key: generate complex implementations of tables by a layering / compositional technology discussed earlier

    - Aside: look at the time frame: 1992 people were not talking about DSLs, barely about program generation, not features and product lines, not software architectures, not design patterns …. were talking about reuse

P2 is DSL Extension to C

- C structure decl:

```
struct employee
{
  int  empno;
  char first_name[20];
  char last_name[20];
  int  age;
  int  class_no;
} employee;              // employee record type
```

- Container declaration was a relation declaration with annotations

```
container <employee>
{

} e1, e2;                // instance declaration
```


Programming with P2

- Declare cursors (iterators) to retrieve tuples that satisfy query

```
cursor <e1> curs where age>35 &&
                       first_name=="Don";
```

  - define cursor “curs” that ranges over container e1 to retrieve only tuples where age>35 and first_name=“Don”

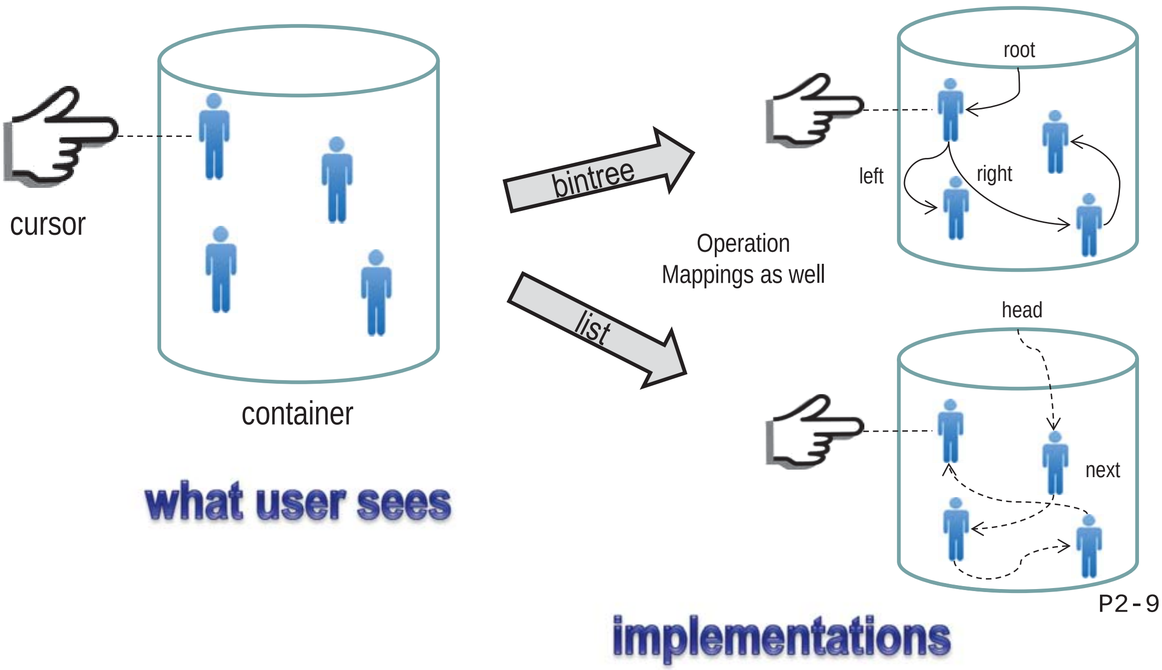
- Iterate over selected tuples with a foreach clause (similar to for-clauses in Java)

```
foreach (curs)
{
  curs.class_no++;
}
```

- Could update, delete tuples during an iteration, insertion, etc. Standard fare…

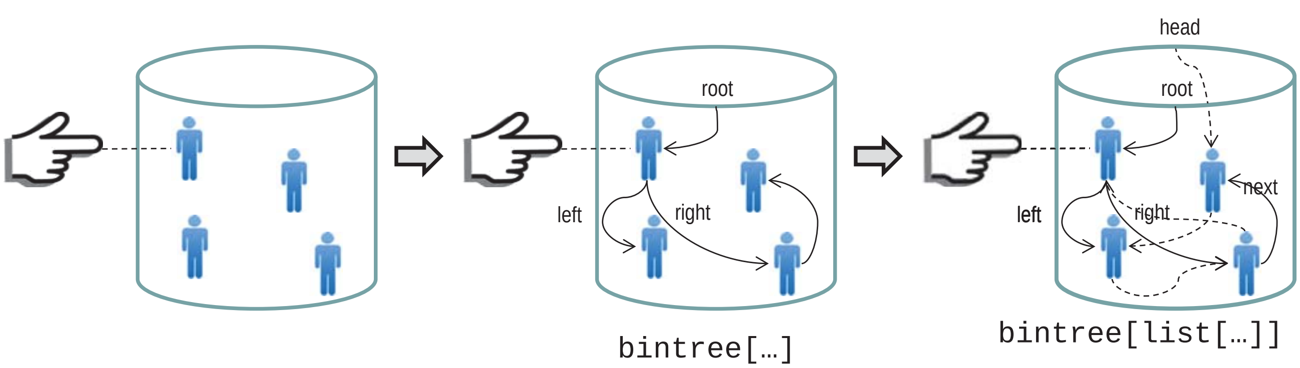
Container Implementations

- Follows prior and general work incremental software development



Novelty of P2 is Map Composition

- Implementations are composed maps



P2 "Maps" or "Components" or "Features"

- Standard data structures were among offered mappings
- Users could compose them to build the customized container data structures that they wanted
- Very much a DBMS-like flavor declaring indexes, file structures, etc .

Layer	meaning
`rbtree`	red-black tree
`list`	linked list
`odlist`	ordered doubly-linked list
`hash`	hash
`df`	delete flag
`hashcomp`	hash compare strings
`predlist`	predicate list
`array`	preallocated array storage
`malloc`	heap storage
`transient`	main memory
`persistent`	on disk

```
typex
{
  list1 = dlist[heap[persistent]];
  list2 = odlist1[odlist2[array[transient]];
}
```

(offhand I don't remember them all – and couldn't quickly find the full list)

Return to P2 Interface

```
list2 = odlist1[odlist2[array[transient]];
```

- "store elements of a container onto a ordered doubly-linked list (`odlist1`) , then onto a second ordered doubly-linked list (`odlist2`) sort keys are different, whose nodes are stored sequentially in an `array` in `transient` memory"
- Container declaration was essentially a relation table declaration with annotations

```
container <employee> stored_as list2 with
{
  odlist1 key is empno; // layer annotations
  odlist2 key is age;
  array size is 100;
} e1, e2;               // instance declaration
```

- Great power to generate huge numbers of customized data structures

Example: Deque

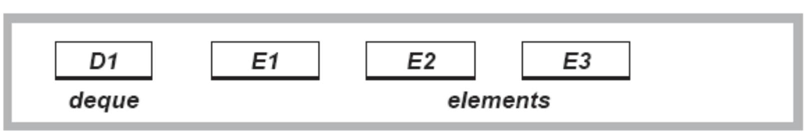
- In the following slides, I show how the code of a Deque (doubly-linked queue) is "derived" to recreate a Booch Library container data structure
    - Look at add_front() method – same for other methods
- P2 uses top-down derivation (generation) of code
    - automated software development by composing layers (mappings)

DeqInterface(Sync(Deque2c(Dlist(Avail(Heap())))))

- Heap() = Malloc(Transient)

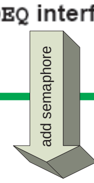
Chain of Mappings

- Start with a DEQ abstraction

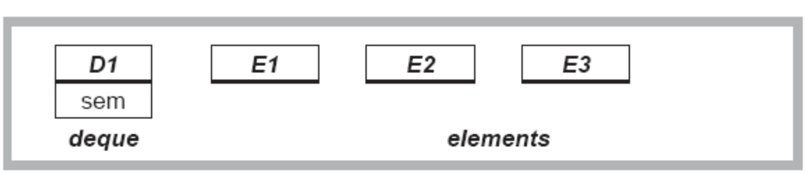



```
add_front (deque d, element e)
{ /* to be defined */ }
```

Figure 1: The abstract objects of the DEQ interface.



- Add synchronous access

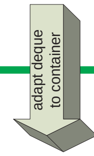



```
add_front (deque d, element e)
{
  wait (d.sem);
  x::add_front (d, e);
  signal (d.sem);
}
```

call layer below (on next page)

Figure 2a and 2b: The deq_sync [x:DEQ] :DEQ mapping.

Chain of Decorators and Adapters



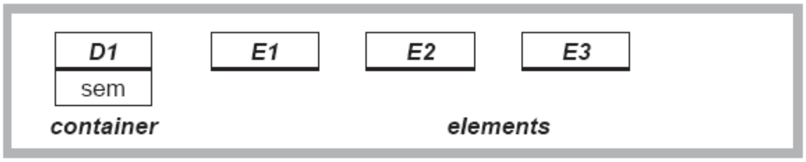
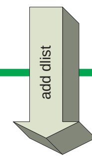


Figure 3a and 3b: The `deque2c[x:CONT]:DEQ` mapping.

```
add_front (d, e)
{
  x::insert_front (d, e);
}
```

call next layer



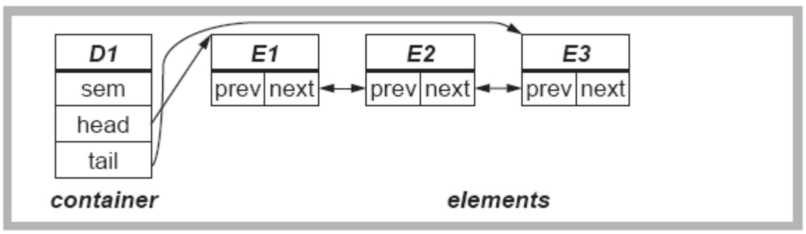
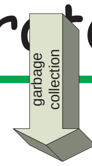


Figure 4a and 4b: The `dlist[x:CONT]:CONT` mapping.

```
insert_front (d, e)
{
  element *g;
  g = x::insert_front (d, e);
  g->prev = NULL;
  if ((g->next = d.head) != NULL)
    g->next->prev = g;
  if (d.head == NULL)
    d.tail = g;
  d.head = g;
  return (g);
}
```

call next layer

Chain of Decorators & Adapters



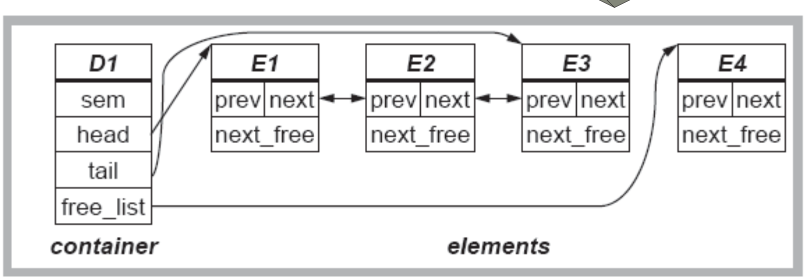
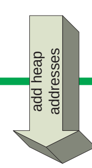


Figure 5a and 5b: The `avail[x:CONT]:CONT` mapping.

```
insert_front (d, e)
{
  element *g;
  if (d.free_list)
  {
    g = d.free_list;
    d.free_list = g->next_free;
    g->data = e;
  }
  else
    g = x::insert_front (d, e);
  return (g);
}
```

call next wrapper



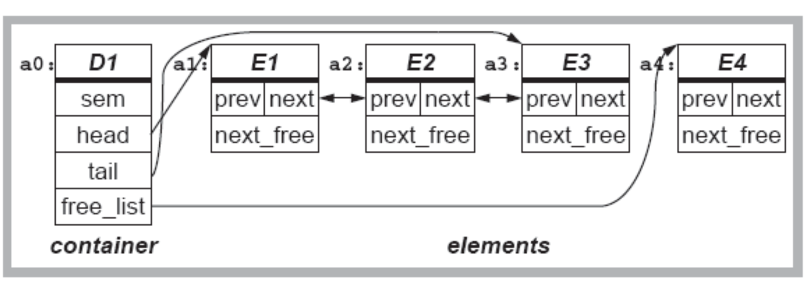


Figure 6a and 6b: The `heap[x:MEM]:CONT` mapping.

```
insert_front (d, e)
{
  element *g;
  g = allocate (sizeof (e));
  g->data = e;
  return (g);
}
```

Macro Expand to Produce Final Result

```
add_front (d: deque, e: element)
{
  element *g;
  wait (d.sem);                        // from deq_sync
  if (d.free_list)                     // from avail
  {                                    // from avail
    g = d.free_list;                   // from avail
    d.free_list= g->next_free;         // from avail
    g->data = e;                       // from avail
  }                                    // from avail
  else                                 // from avail
  {                                    // from heap
    g = malloc (sizeof (e));           // from transient
    g->data = e;                       // from heap
  }                                    // from heap
  g->prev = NULL;                      // from dlist
  if ((g->next = d.head) != NULL)      // from dlist
    g->next->prev = g;                 // from dlist
  if (d.head == NULL)                  // from dlist
    d.tail = g;                        // from dlist
  d.head = g;                          // from dlist
  signal (d.sem);                      // from deq_sync
}
```

Figure 7: The Composed Mapping of `add_front()`

Paper Title:
Scalable Software Libraries (Sigsoft 1993)

- Because we could create huge number of distinct deque and other container implementations: grammar of the language of compositions, each sentence defines a unique composition (data structure)

```
Deque  : deque2c  Cont
       | deq_sync Deque
       ;

Cont   : dlist    Cont
       | odlist   Cont
       | array    Mem
       | heap     Mem
       | avail    Cont
       | bintree  Cont
       | redblack Cont
       ;

Mem    : persistent
       | transient
       ;
```

could have replicas: multiple odlists, bintrees, etc.

Performance (No Contest)

- Less code to write because of declarative specs, typically faster code because of obvious optimizations that could be performed – such as query optimizations

Component library	Unordered linked list	Unordered array	Sorted array	Binary tree
Booch C++ Components 2.0-beta	320 words	360 words	398 words	481 words
libg++ 2.4	336 words	386 words	474 words	336 words
P1	281 words	281 words	287 words	285 words
P2	308 words	310 words	316 words	310 words

Table 1: Code size of dictionary benchmark programs (in words of code).

Component library	Unordered linked list	Unordered array	Sorted array	Binary tree
Booch C++ Components 2.0-beta (compiled with Sun CC 3.0.1 -O4)	70.9 sec	54.6 sec	11.1 sec	15.4 sec
libg++ 2.4 (compiled with G++ 2.4.5 -O2)	41.9 sec	34.3 sec	5.4 sec	4.1 sec
P1 (compiled with GCC 2.4.5 -O2)	40.2 sec	33.3 sec	6.3 sec	3.0 sec
P2 (compiled with GCC 2.4.5 -O2)	40.3 sec	33.3 sec	6.2 sec	3.2 sec

Table 2: Running times of dictionary benchmark programs (combined user and system time).

Query Optimization

- Consider the following predicate: **name == "Don" and age<20**
  - name is primary key

- Suppose 2 data structures are superimposed:
  - unordered list
  - ordered list on ascending age

- Unordered list layer would analyze the predicate and produce the approximate code below, which on average searches ½ of the elements

```
foreach( element : unorderedList ) {
     if (element.name=="Don") {
        if (element.age<20) return element;
        else return null;
}
return null;
```

Query Optimization (Continued)

- The age-ordered list would produce the following code, searching on average ¼ of the container

```
foreach( element : orderedList ) {
     if (element.age<20) {
        if (element.name==“Don”) return element;
        else continue;
     }
     return null;
}
```

- Each layer is “queried” for its cost estimate – the layer that responds with the lowest cost is the data structure that is traversed to produce qualified tuples
- Standard fare in RDBMS implementation

LEAPS Example (Sigsoft 1994)

- Production system compiler that produces fastest executables of OPS5 rule sets
- Used very complicated container data structures
- We re-engineered LEAPS called RL, showed P2 scaled to this complex application, and produced unexpected performance gains

- Used standard LEAPS benchmarks

Rule Set	Rule Set Size	RL-generated P2 Program Size	P2-generated C Program Size	LEAPS-generated C Program Size
`manners`	8 rules	770 lines	3,300 lines	2,300 + 10,000 lines run-time
`waltz`	33 rules	2,400 lines	13,600 lines	10,000 + 10,000 lines run-time
`waltzdb`	38 rules	3,100 lines	15,800 lines	15,000 + 10,000 lines run-time

Table 1: Size of Generated Programs

- Performance of P2 code about 1.5-2.5 times **faster** than LEAPS executables

Unexpected Benefits

- We understood what they were doing
- We could easily swap data structures (in this case container join algorithms) and have run-times that were orders of magnitude faster than LEAPS

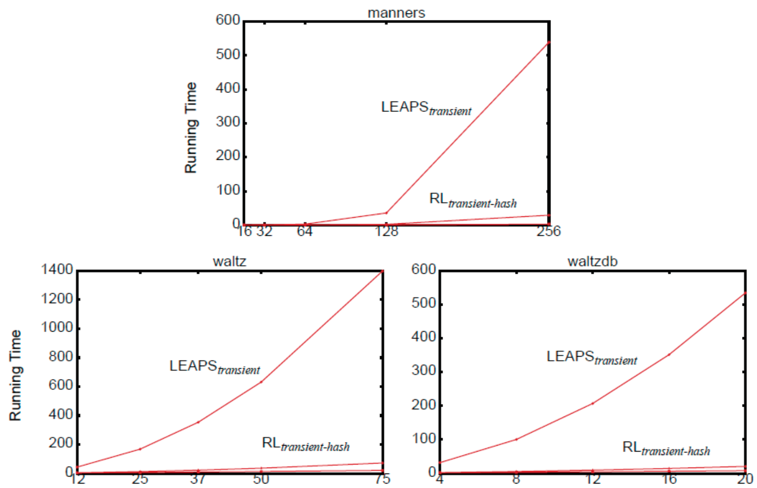


Figure 11: LEAPS$_{transient}$ and RL$_{transient\text{-}hash}$ execution times, showing LEAPS$_{transient}$ using nested loops join versus RL$_{transient\text{-}hash}$ using 13 and 1011 hash buckets.

Unexpected Benefits

- P2 encouraged the exploration of different designs & data structures
- LEAPS compiler is 20K LOC; our RL is 4K LOC
  (productivity increase $\times 3$, code volume $\times 4$)
- Only had to write 2 additional layers to generate LEAPS data structures
- P2 enabled novices (ourselves) to program like domain experts

LESSONS LEARNED IN HINDSIGHT

To The Participants of This Dagstuhl

- Design is more than a SAT problem
    - Not difficult to find a feasible solution – lots of crappy implementations
    - Hard part is to find an efficient implementation
    - Ex: Relational Query Optimization

Lessons Learned #1

- Thank Ras…
- Periodically, I see a resurgence in this kind of work (PLDI'11 "An Introduction to Data Representation Synthesis"), but often people aren't aware of prior work
  - how do we know what problems to work on if we don't know what already has been done?
  - who would know except people with long memories?
  - no standard lexicon or terminology – can't find my work or that of others by googling "Data Representation Synthesis" or "Data Structure Synthesis"
  - Actually, I couldn't find it myself ☹
- Consequence – results and knowledge disappear
  - not clear if they are reinvented

Lessons Learned #2

- Relying on referees to know prior work (particularly areas whose core knowledge crosses many years) is increasingly a bad idea
  - referees generally don't know the literature in automated program development and generally don't care – at present it is a limited audience
  - consequence: referees are unaware of when useful or significant progress is being made
- Dewayne Perry's observations on typical Software Engineering referee reviews:
  - "you didn't do it my way"
  - "that's not the way that I would do it"

Lessons Learned #3

- Automated Program Development $\supseteq$ 'program synthesis' is a difficult and fundamental problem that will take years or decades to become mainstream

- Trend in SE research is: if you have a better way X than Y to solve problem P, you must implement both solutions and have sound experimental results to demonstrate how X is better than Y
  - in program automation, this is often the wrong question
  - we are more at a stage: is it possible to solve automatically?
  - deal with large or complex programs, such experiments may be infeasible

- In an area that begs for abstractions and grander theories, the bar is being raised beyond reach for such ideas to be published in visible venues
  - research programs that are long-term will have difficulty surviving today's conditions

Lessons Learned #4

- Arguably the greatest single result in automated software development is

relational query optimization

- Yet, pick any undergraduate text on software engineering in the last 20 years. I defy you to find a reference to RQO
  - RQO about 30+ years old
  - revolutionized a fundamental branch of computer science
  - and whose results (program designs are algebraic expressions that can be optimized) and relevance to automated design is unknown, unappreciated, or forgotten

What are we teaching our students?

Lessons Learned #5

- Consequence: every new generation of students (and faculty?) has no idea of what has been done
- **So how do we know what to work on if we don't know what has been done?**
- Ans: we don't. That's why we have Dagstuhl Seminars....☺

Thank You!

- Questions?