Outrageous Software

John Hamer

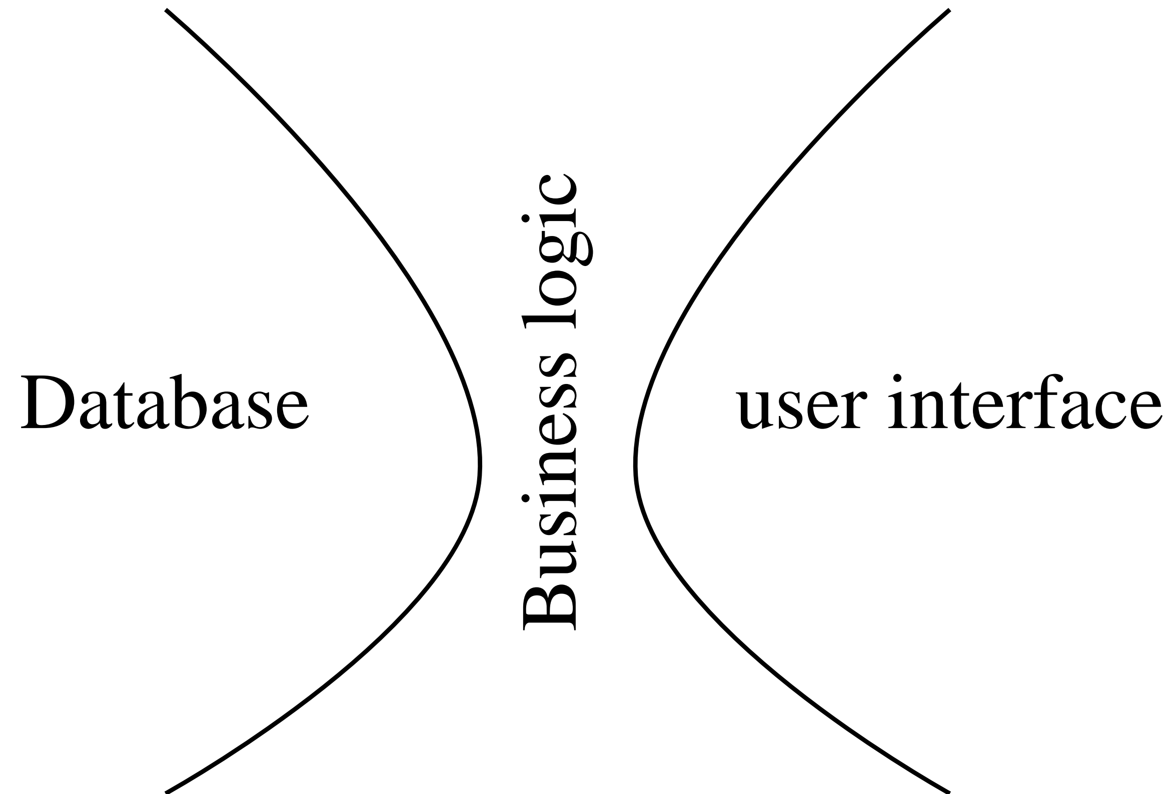
J.Hamer@cs.auckland.ac.nz

“Today, most software exists, not to solve a problem, but to interface with other software.” —I.O. Angell

“Perilous to us all are the devices of an art deeper than we possess ourselves.” —J. R. R. Tolkien

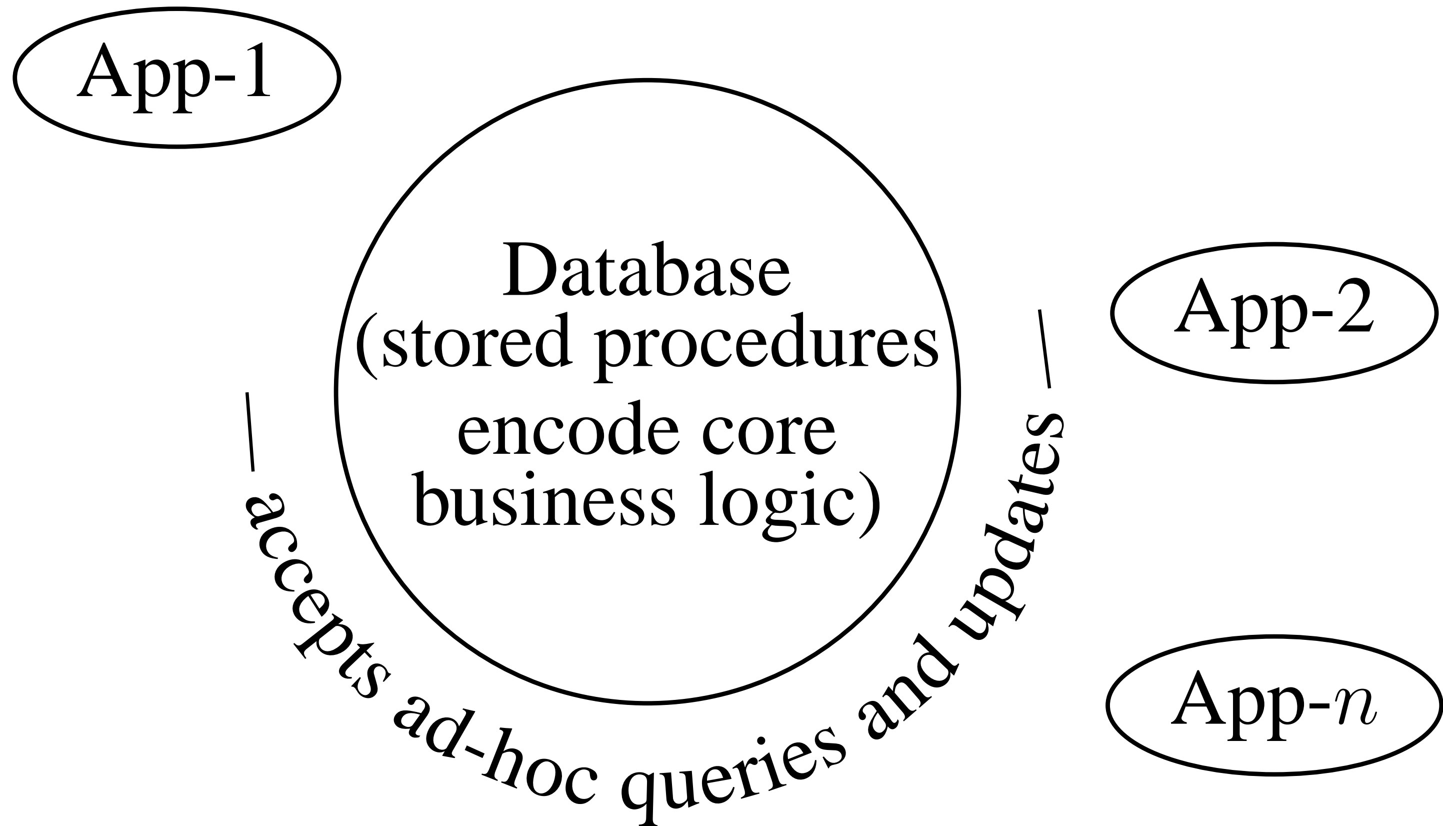
“Out of intense complexities, intense simplicities emerge.” —Winston Churchill

An enduring meta-architecture



- May be monolithic, client-server, web service, etc.

Codd's meta-architecture



- Domain model is *relational* and entwined with database

The sad reality

- Database logic contaminates the business logic
- User interface logic contaminates the business logic
- Business logic contaminates the database logic
- Business logic contaminates the user interface

Database and U.I. independence is *not* the issue

- *Database independence* means using generic (ANSI) SQL, together with an abstraction library (to handle, e.g., placeholder syntax)
- *U.I. independence* means using a cross-platform library, writing facades ("application layer"), etc.

Contamination is a failure to separate concerns.

- E.g., foreign keys appear in objects; logic for retrieving data is mixed with business policy logic; …
- All this conspires to diffuse and complicate the business logic.

Impedance mismatch

- The relational data model is fundamentally incompatible with the object-oriented model
- famous “impedance mismatch”:
  - OOP: traverse objects via their relationships; relational: join data rows of tables.
  - OOP: model includes behaviour and data; relational includes only data.
  - OOP: keys typically not shown; relational: explicit keys.
- Similar “impedance mismatch” occurs between the domain model and the user interface.
- In addition, user interfaces are diverse and highly changeable.

When you’re in a hole, dig deeper

- Current solutions all involve adding more abstraction layers; e.g., frameworks like EJB and .Net, JDO, etc.
- Popular IDEs for creating user interface components by mapping databases tables
- The underlying mismatch remains.

“I have yet to see any problem, however complicated, which, when you looked at it in the right way, did not become still more complicated.”

— Poul Anderson

“Outrageous Software”

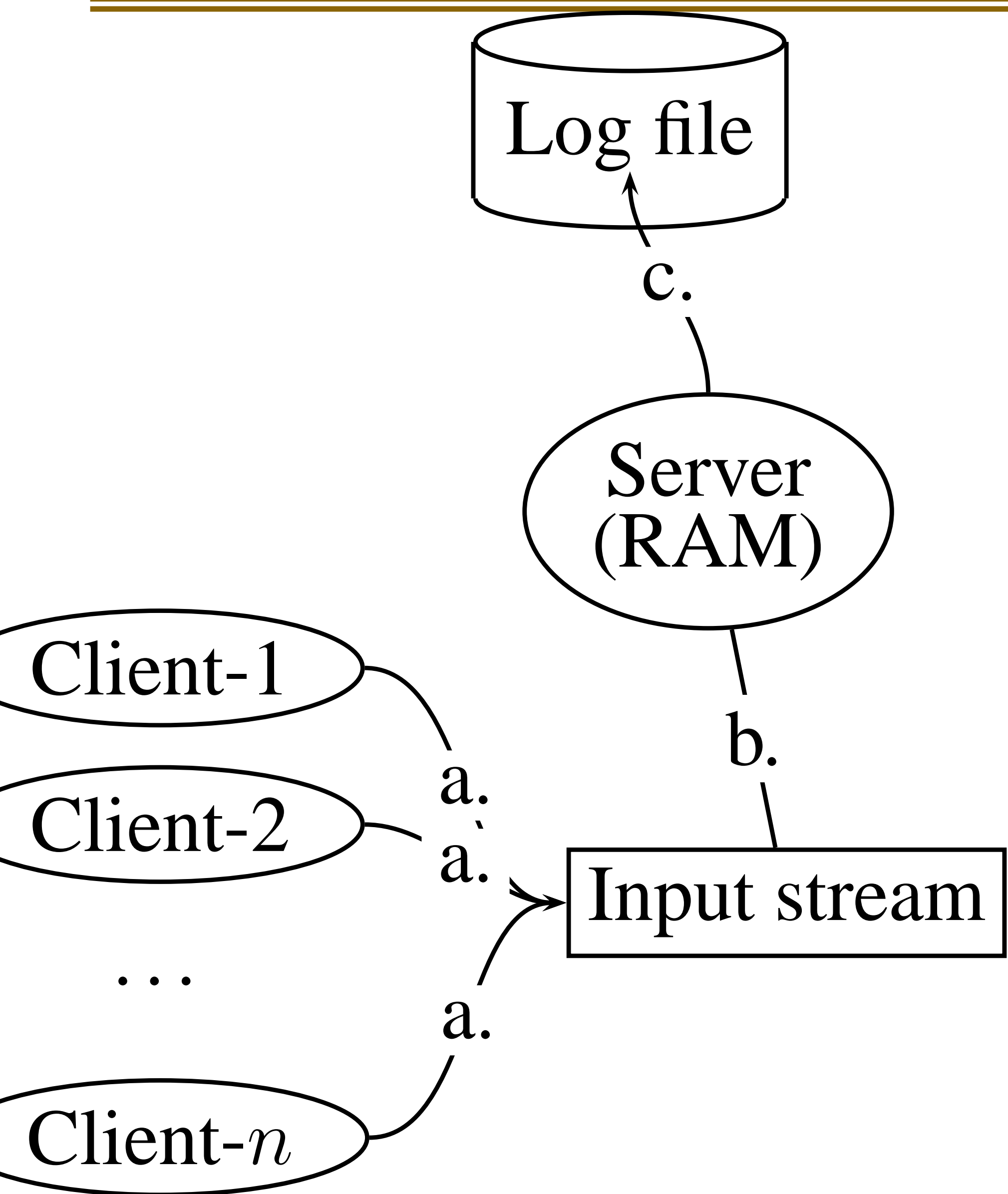
Combination of three radical software technologies

prevaylance replaces conventional relational databases, providing a pure object model, extreme performance, and adaptable fault-tolerant distribution; `http://www.prevayler.org`

naked objects eliminate the need for hand-crafted user interfaces; `http://www.nakedobjects.org/`

Mozart/Oz a multi-paradigm programming language supporting massive concurrency, transparent distribution, and an integrated finite domain constraint solver; `http://www.mozart-oz.org`

Prevaylance: trivial persistence



- Client places a command in
  serialized input stream (a.)
- Server reads command (b.),
- logs the command (c.),
- then executes.
- Log file provides persistenc
- to restore, for each log entry
  set date-time, execute.
- Occasional "snapshot
  (memory dumps) minimi
  restore times.

Features

Very, *very* fast orders of magnitude faster than *cached* relational database.

Trivial implementation e.g., a few hundred lines of Java.

Language independent Java, C♯, C++, Perl, Python, ...

Only minor impact on programmer some conventions to follow for serializing commands.

Distributed architectures e.g., fire up a query server on any machine that can read the log file; or, hot-swap main server by redirecting input stream.

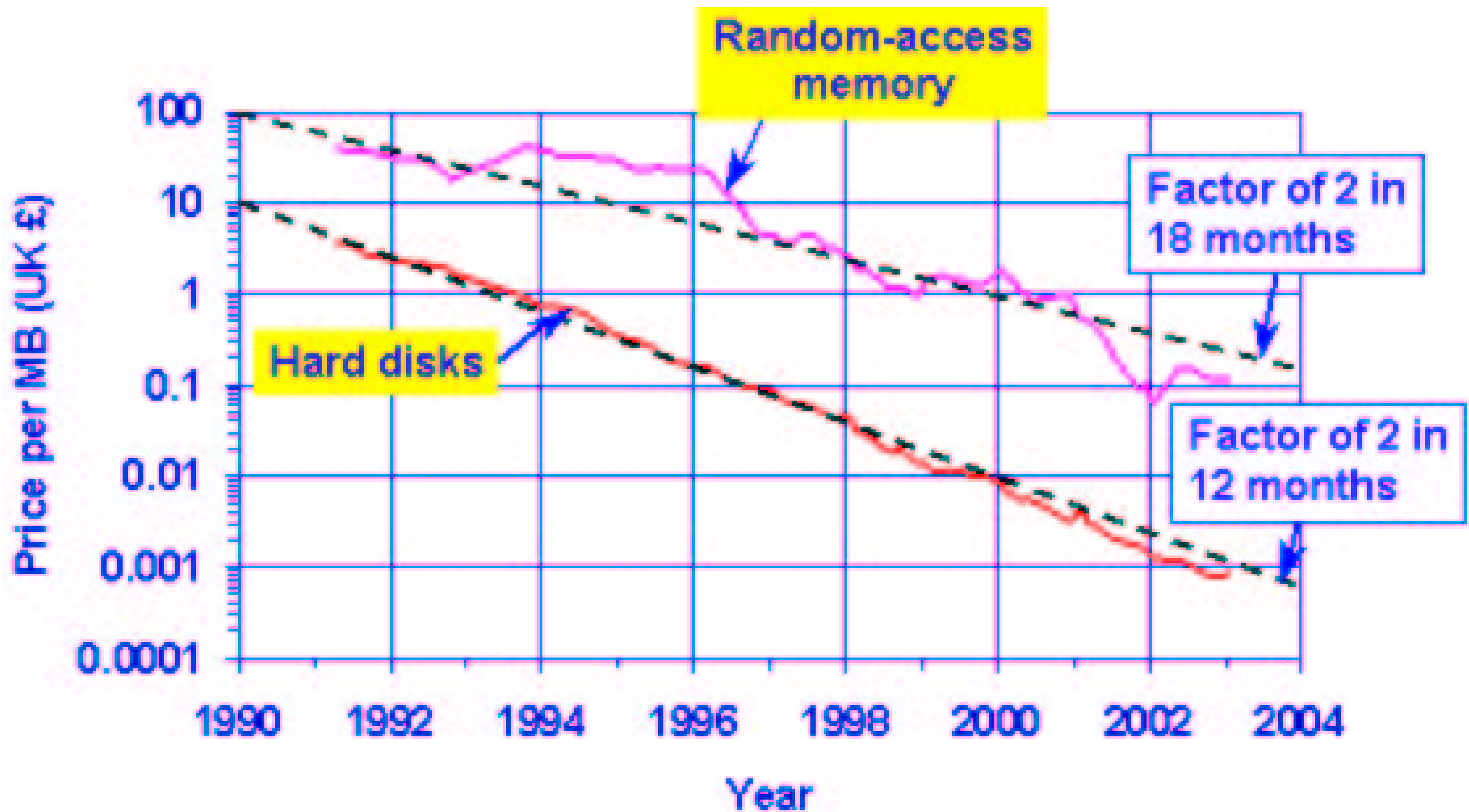
And it actually works!

Store all active data in RAM

Premise: your server has sufficient RAM to hold the complete set of business objects.

- 64-bit address space will be adequate for "some time to come"
- RAM may be 100 times more expensive than disk, but *business data is not growing at an exponential rate*.
- $-per-Mb for RAM in 2004 is roughly equivalent to $-per-Mb for disk in 1997.
- Data tends to be stored more compactly in RAM than disk.

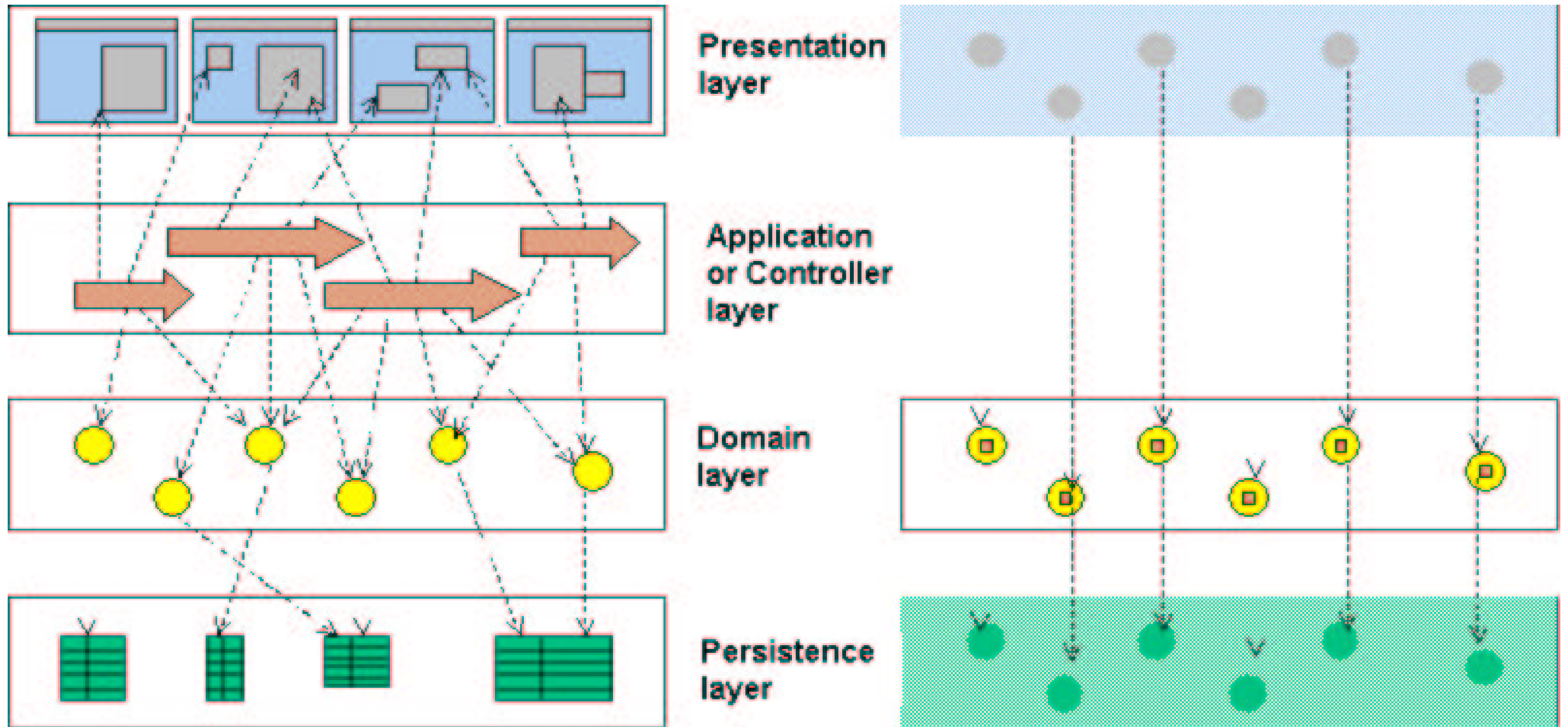
From EUB Technical Review, April 2003



Naked Objects

- Domain objects are then rendered directly visible to, and manipulable by, the user.
- All user actions consist of creating or retrieving objects, specifying their attributes, forming associations between them, or invoking methods on an object (or collection of objects)

Naked Object Architecture



What’s left: just the meat

- The combination of prevaylance and Naked Objects immediately reduces the coding required for business software (est. 60% or more), while simultaneously eliminating the distorting effects of relational database and user-interface logic.
- Focus can now be placed on the business logic.

So many business systems. Are businesses really so diverse?

Generic business logic

- Not a new idea: IBM's SanFrancisco framework, Hay's Data Model Patterns, Fowler's Analysis Patterns, GNUe project, ...
- E.g., Financials, Supply Chain Management, Customer Relationship Management, Human Resources, Project Management, ...
- Not without problems
  - Terminology varies between industries
  - Tends towards "overkill" — all possible features
  - Businesses want to preserve their own "distinctive character"

The “Generic-Specific” approach

- Claim: Generic models are best defined in abstract, mathematical terms (graphs, sets, relations, etc.)
- Allow formal properties to be established (you don’t want the GL to leak $…)
- Constraints can be written in a declarative language that admits formal reasoning.
- Propose *instantiation phase* that introduces client terminology and structural constraints.
- Deployed system is a *specialised* business model, not intended for reuse.

Example: *Accountability*

- Concerns "who reports what to whom." Structures are typically hierarchical (but not always).
- E.g., sales office reports to division, division reports to region, region reports to group.
- There may be several dimensions of accountability, such as: product support, sales, etc.

Generic accountability model

- A *generic* model of accountability must accommodate all such arrangements; e.g., a labelled directed acyclic graph, the nodes denoting roles and the edges denoting the responsibility.
- *Instantiating* involves attaching constraints on the graph.
- Graph nodes may be classified into various categories or roles (secretary, line manager, etc.).
- Note that after instantiation, there is still no "active" data in the system; e.g., the individuals who fill the various roles are not known. The instantiated model serves to (a) constrain the data that is subsequently entered; and (b) allow "compilation" to an optimised system.

Mozart/Oz

- Multi-paradigm language (dataflow, functional, logic, object-oriented, active objects, distributed state, constraint, component-based)
- Small “kernel” with rigorous formal semantics.
- Outcome of 10+ years programming language research
- Attention to pragmatics that is unusual in an “academic” language.
- Ideal substrate for Outrageous Software.

Constraint problems

- In an “algorithmic” problem, the steps to reach a solution are known (and fixed). You are familiar with these.
- In a constraint problem, the steps are not known. All that is known is the problem structure and the *properties* a solution must hold. A solution is found using *search*.
- E.g., scheduling, warehouse allocation, product configuration, etc.

Constraint problems (cont.)

- “Algorithmic” solutions tend to be difficult to derive and are (by their nature) inflexible.
- Traditional programming languages offer *no support whatsoever* for constraint solving.
- Tend to adopt crude algorithmic approximations, or avoid the “tricky bits” altogether.

Constraint solvers

- Most search problems are “combinatorial”; i.e., exploring all states is impractical.
- To be effective, constraint solvers apply a variety of search strategies.
- Mozart/Oz provides mechanisms for controlling the search strategy, and to facilitate communication between sub-goals.
- Relies on support for (massive) concurrency with synchronisation through dataflow variables.

Reality check

- No commercial support for Mozart/Oz (use Java, but lose the constraints)
- 64-bit language implementations not yet mature? (in a year or two)
- No support for ad-hoc queries (work in progress)
- Likewise for interoperability (XML is replacing SQL as the *lingua franca*)
- To radical to gain traction (become a radical).

Summary and conclusions

- True OOP is not possible in conjunction with non-OOP relational and user-interface layers.
- Massive efforts currently underway in interfacing across the divide (EJB, JDO, .Net, IDEs, . . .)
- These “awkward ends” can be eliminated using some alarmingly simple technology.
- Focus can then (finally!) turn to the heart of the matter: the core business logic.

Summary and conclusions (cont.)

- Reusable business logic requires generic models.
- These should be mathematically based, to allow formal properties to be established.
- *Instantiation* step necessary to bridge terminology gap and bind constraints.
- Changes the nature of programming: become formal modellers.