Reuse: Dealing With The Hand You're Dealt

Jonathan Haulund
Chief Software Engineer
AEHF Program Office
United States Air Force

Why Reuse?

- Definition of software reuse: Check Wikipedia
  - Let’s focus on extant code reuse – extending software from an existing system

So why?

- “Cheaper” – leverage investment
- “Quicker” – not starting from scratch
- Ease adoption by user community (maintains known business model and operational concepts)
- Large, complex and esoteric implementation with a proven track record… why not?

Technology Magic

- Any sufficiently advanced technology is indistinguishable from magic.
  - Arthur C. Clarke, "Profiles of The Future", 1961 (Clarke's third law)
- SOA – “Let’s wrap it!”, seamless integration of legacy systems…
- OOP – encapsulation, inheritance, composition, reusable objects…
- ODBC, JDBC, JCA…

Feasibility Study

- Spend time and money before you waste time and money
  - Analyze the reuse architecture/design thoroughly in the context of the new product requirements
  - Prototype concepts of how the code will be incorporated into the new product, how the code will be extended to meet new requirements
  - Ask the questions: “Does this reuse impose design limitations I can (can’t) live with?”, or rather, “How do I reuse this code without imposing painful design limitations?”
  - Consider testability, extensibility and maintainability
  - Set threshold for code refactoring i.e. what percentage of the existing code base is going to change?

“Real World”

- Many large legacy systems are often built over time and tend to evolve rather than follow a deliberate design process
- Prototypes and engineering software become commercial/operational products
- Sometimes the best technical approach is not considered the best approach. There are other factors that influence the decision to reuse

  e.g. cost, schedule and community buy-in

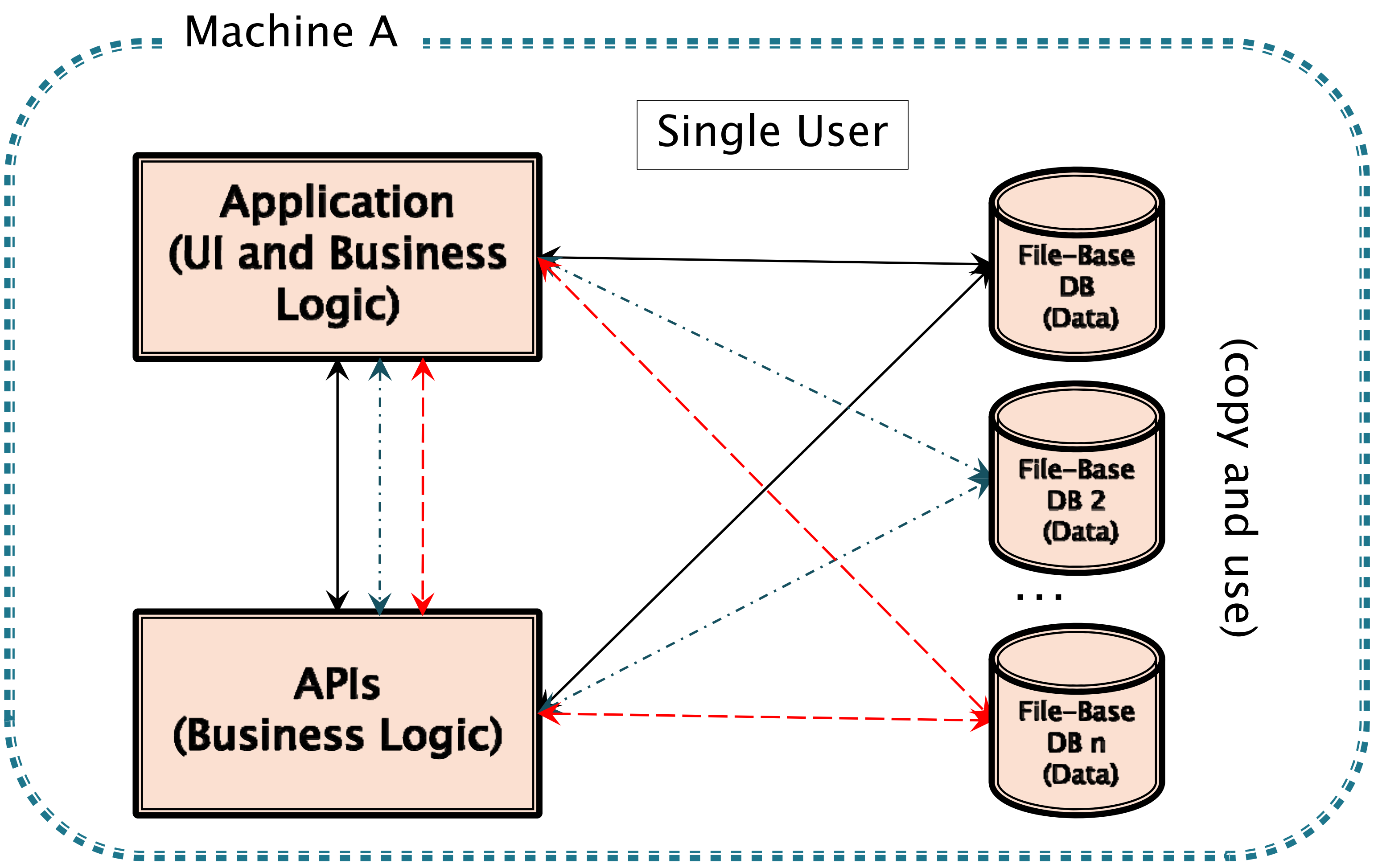
The Good, The Bad & The Ugly

- The Good
  - Modular, loose coupling, well documented (code comments & design artifacts), obvious and logical design patterns, highly testable, solid data model, easily extensible, etc.
- The Bad
  - Opposite of The Good
- The Ugly
  - Good enough to attempt but there will be pain i.e. deal with the hand you’re dealt

For Example…

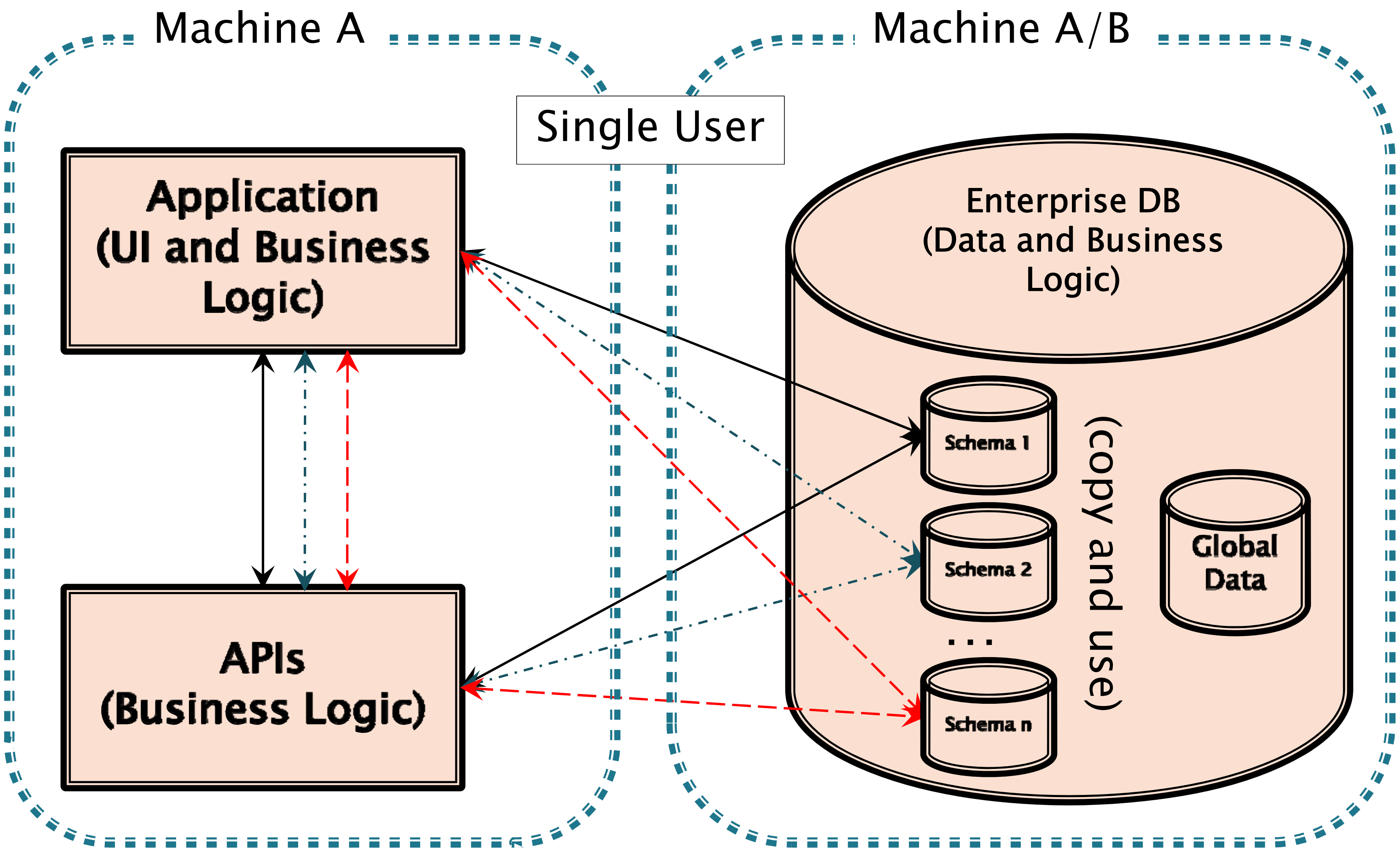
- Progenitor
  - <u>Single User</u>, <u>1 Tier</u>, <u>File-based DB</u>, <u>Legacy Proprietary Development Libraries</u>
- Reuse 1
  - Single User, <u>2 Tier</u>, <u>Enterprise DB</u>, Legacy Proprietary Development Libraries
- Reuse 2
  - <u>Multi User</u>, <u>3+ Tier</u>, <u>Service Oriented</u>, Enterprise DB, Legacy Proprietary Development Libraries

Progenitor


Machine A
Single User
Application (UI and Business Logic)
APIs (Business Logic)
File–Base DB (Data)
File–Base DB 2 (Data)
. . .
File–Base DB n (Data)
(copy and use)

Reuse 1

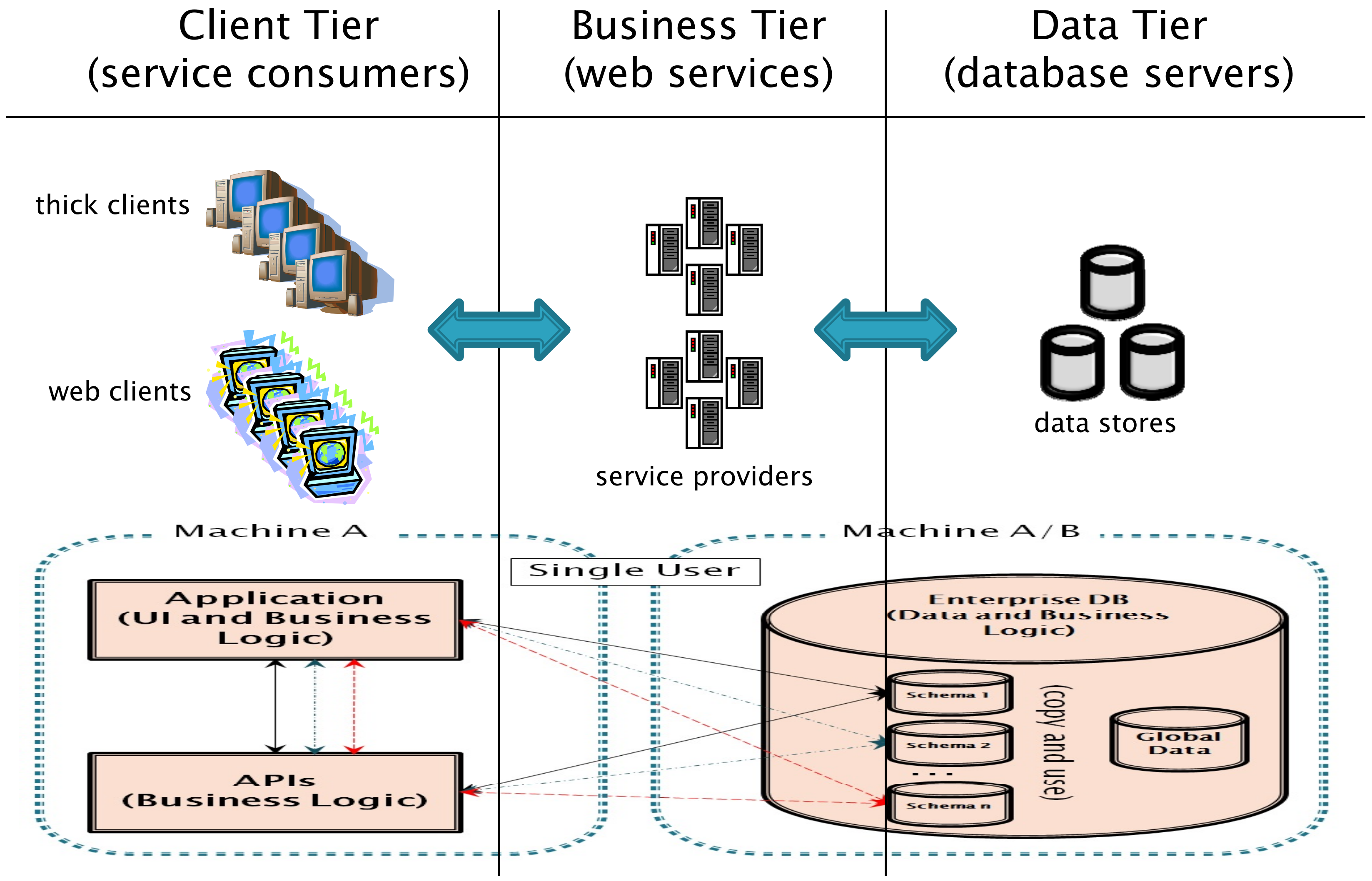
Retains tight coupling of UI, BL and DB



Schemas replace files – reduces refactor, retains single user

Reuse 2

Tight coupling of UI, Business Logic, and DB



Data model only supports single user, will not scale

Final Thoughts

- Sometimes the best technical approach is not considered the best approach
- A feasibility study is a good idea
- Ask the question: “How do I reuse this code without imposing painful design limitations, both now and in the future?”
- Make extensibility, testability and maintainability a major part of your reuse strategy