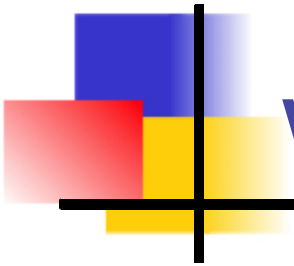
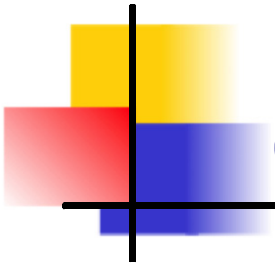


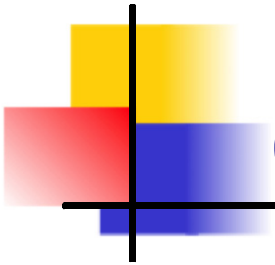
What OO Doesn't Address

Simon Roberts
simon@dancingcloudservices.com



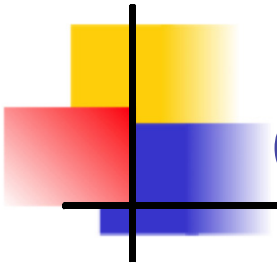
Object Orientation

- Why do we do OO
  - There are other ways, though far less popular now
  - Until the mid '80s OO was largely an academic curiosity.
- “Pure” OO isn't the answer to everything!
  - To know when to cheat, you must understand why the rules exist



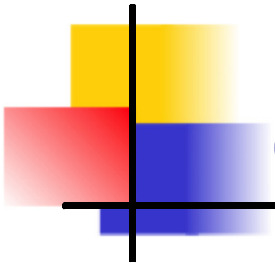
Complexity

- In 1975 Yourdon described 100,000 to 1,000,000 lines of source as “nearly impossible” and more than 1,000,000 as “utterly absurd”
- Now this magnitude is almost commonplace



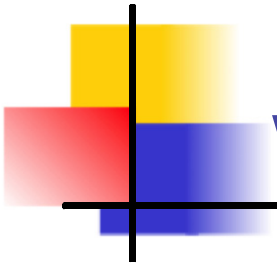
OO Promises

- OO made many promises when first adopted by mainstream commercial fields
  - Reuse would ensure you never write the same thing twice
  - That reuse would amount to re-testing, so quality would soar
- We do get much better reuse, particularly in library code
  - But many “Customer” classes exist!



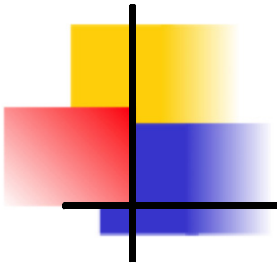
OO Addresses Maintenance

- Despite some campaign promises, OO stayed because code quality, reliability, maintainability improved.
  - Maintenance starts with the second line of code, therefore OO improves “writeability”
  - Maintenance typically accounts for 90% of the total project cost



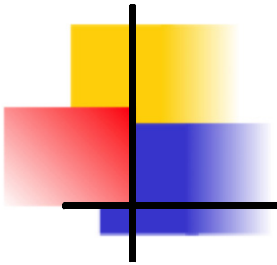
Where To Start?

- Two broad ideas:
  - Limit the consequences of change
    - Resilience not resistance
    - Change can occur at design time or runtime
  - Write code that's Grokable with fullness
    - Understand causes, consequences, and where to look in your code
    - “Grok means to understand so thoroughly that the observer becomes a part of the observed—to merge, blend, intermarry, lose identity in group experience” R.A. Heinlein 1961



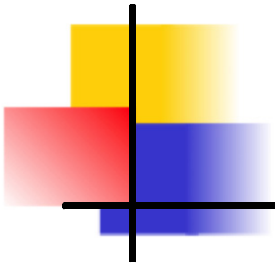
What's Volatile, What's Not?

- Would you use rock to build a house on mud and rock slurry, or mud and rock slurry to build a house on rock?
- Some aspects of a system are relatively stable, some less so
  - Customer
  - Invoice workflow
- Consider procedural programming
  - Great if you're a mathematician



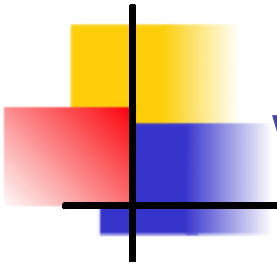
How Should An Object Look?

- What if you design an object to serve its user?
  - It's probably efficient
  - It's probably easy to use — for that user
  - What if the users, or their priorities, change?



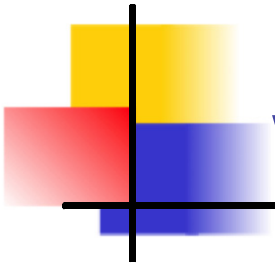
How Should An Object Look?

- What if you design it to be “true to itself”?
  - Can serve any legitimate use — even if the use changes
  - Might not be so efficient or convenient



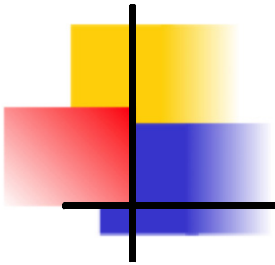
What's Inside An Object?

- None of your business!
- Encapsulation of data reduces dependencies/coupling
  - Now, if the innards change, you don't know or care
- These ideas create a boundary limiting consequences of change
  - From the inside, no effect on the outside
  - From the outside, no effect on the inside



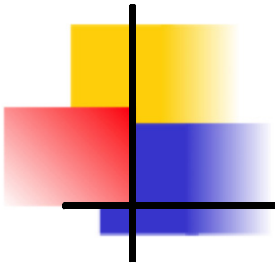
What's Part Of An Object?

- Consider Person<-Employee<-Manager
- Consider BankAccount
  - Which GoF design pattern exemplifies this?
- Keep apart things that change independently
  - So design changes in one part don't break another part
  - So runtime changes are supported



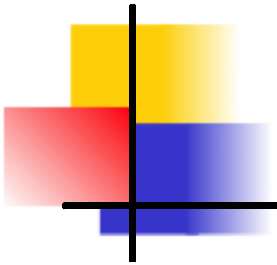
Understandable Ideas

- Base your model on well understood ideas
  - Customer, Invoice, Strangeness, Charm
  - Not abstract stuff you came up with to solve the problem
  - Now any team member should have a clear, and correct, expectation of roles and responsibilities
  - Suggests that design should be done by people experienced in the business domain



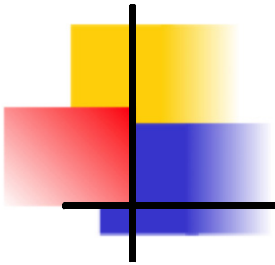
Understandable Ideas

- Objects/classes should pass the “duck test”
  - So you know where to look when things change or need fixing
  - Ties in with creating objects that are “true to themselves”
- Individual methods should pass duck test too
  - And nothing more



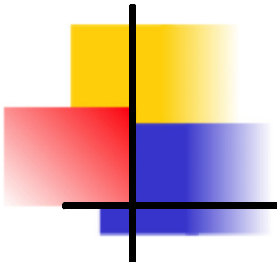
Know What To Blame

- Errors should pass the “duck poop” test
- When you find Feb 31st lying around your program, where is the bug?
  - In 1,000,000 line procedural program “it's in the program”
  - In 1,000,000 line OO program, “it's in the date class” — because that's the only code that can change a date if it's properly encapsulated



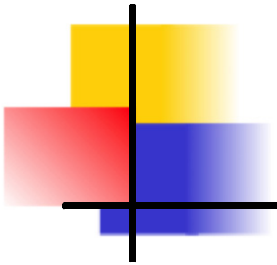
Know Sooner Not Later

- Appropriate use of exceptions tells us immediately an error arises
  - Pre-conditions
  - Post-conditions
  - Invariants
- Don't keep walking if there's duck poop on the ground



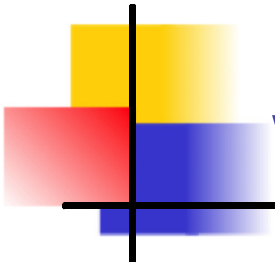
Recap

- Keep related things together and unrelated things apart
- Keep things that change independently apart
- (Keep things that change together together?)



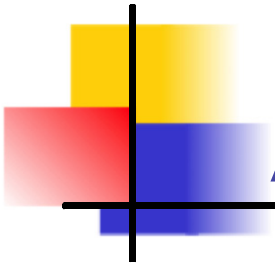
Recap

- Base model on stable aspects
- Objects should model well-understood aspects
- Classes should be like Hollywood actors
- Design without regard to use
- Encapsulate
- Apportion blame, know immediately when trouble arises



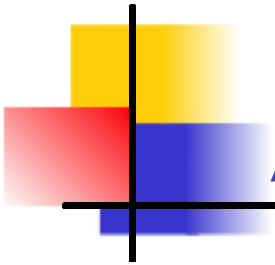
What Have We Got?

- Code we understand
  - We can fix it
  - We can add to it
  - We can reuse (some) parts of it
- Improvements in correctness
- Flexibility, maintainability, extensibility
- Testability



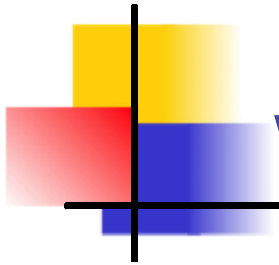
An Observation About OO

- Consider the type of machine that OO "assumes"
- It's perfect; mathematical perfection
- Consider:
  - Speed
  - Memory
  - Power-requirement
  - Correctness



About Perfect Computers

- Infinitely fast; concurrency isn't a meaningful consideration even if threads exist
- No memory limits, no failures, no power, no on-off switch
  - Therefore, no need for a database
- If there's a network, it too must be infinitely fast, secure, and never fails
- OO ignores network, concurrency, persistence, performance



Why Do Projects Fail?

Bad/Changing Requirements

Budget Overruns

Key People Leave

Security

Usability

Reliability

Time Constraints

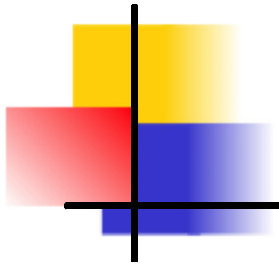
Scalability/Capacity

Performance

Availability

et cetera, et cetera

- Which of these does OO address?



Why Do Projects Fail?

Bad/Changing Requirements

Budget Overruns

Key People Leave

Security

Usability

Reliability

Time Constraints

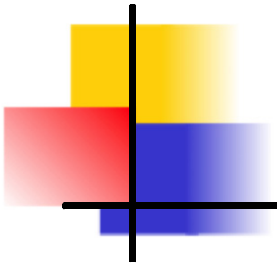
Scalability/Capacity

Performance

Availability

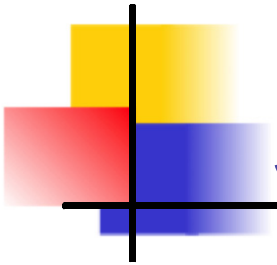
et cetera, et cetera

- Which of these does OO address?



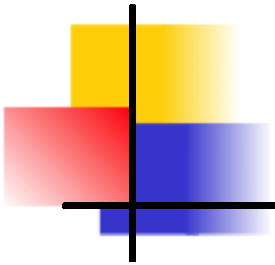
What About Performance?

- Guidelines have not been about speed
  - Lots of extra communication — compare with large company organization
  - No optimization to suit use
- OO code is typically ~20% slower than procedural
  - Partly explains why adoption was delayed till the '80s



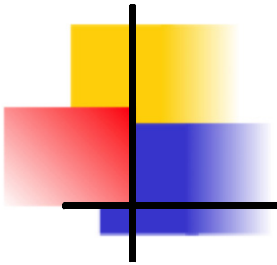
Speed

- Once upon a time, performance was about good algorithms
- Today much of response time is made up of network latency
  - OO tells us to ignore how something is used
  - Might well lead to poor network behavior and unacceptable response time



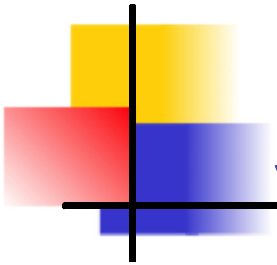
Network Behavior

- Latency
  - Minimize round-trips
- Bandwidth
  - Avoid sending unnecessary data
- What about partial failure?
- What about the zeroth law?



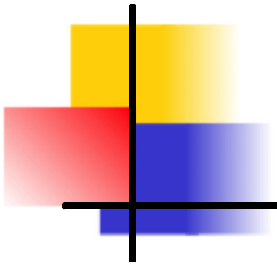
Reasons For Networking

- Access remote resources
  - Users, data, processes, compute horsepower...
- Clustering
  - Capacity, failover



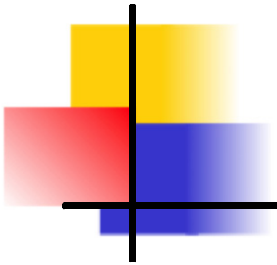
Scalability

- Hard limits on scalability can arise because of Amdahl's law
  - Transactions are a compelling example
- Our OO outline did not take any notice of concurrent or serial execution
  - Default behavior, particularly spread over a network, might be catastrophic for scalability
- Avoid hard transactions if possible, and keep essential ones short, and local



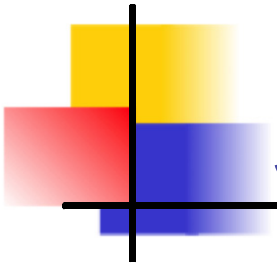
Reliability

- In the sense of “transactional correctness” rather than availability
  - Start with N^{th} normal form
  - Denormalize for good performance—based on the way the data are used
  - But how did we design our objects?
- Use a thoughtful approach to ORM



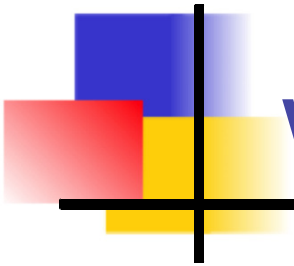
Memory

- The only weakness of Java (or more probably any garbage collected system) is incompatibility with virtual memory
  - OO doesn't address memory use
  - Java can be quite sensitive to object lifetime
- Beware of pooling
- Beware of objects of medium lifespan



Summary

- OO isn't the be-all and end-all, but is much better than previous approaches
- OO addresses maintenance
  - Limit consequences of change
  - Really understandable code
- Network, transactions, persistence and other issues must be considered too
  - These are commonly referred to as “architectural”



What OO Doesn't Address

Simon Roberts
simon@dancingcloudservices.com