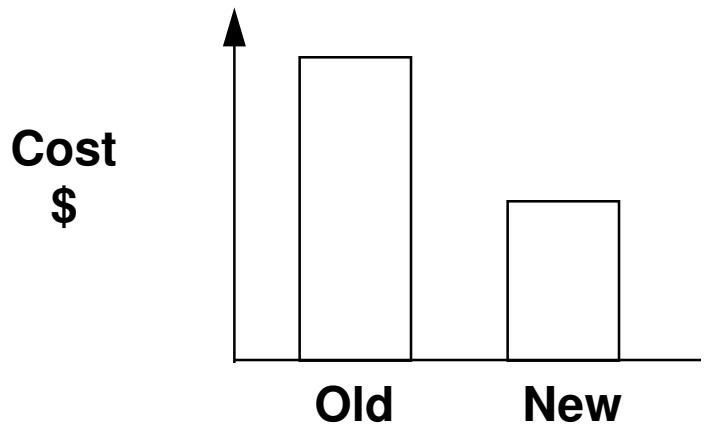
OBJECT-ORIENTED DESIGN
THE NEED

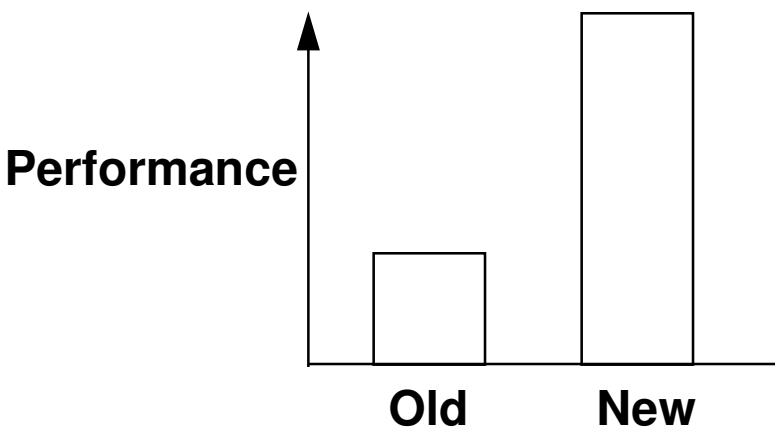
- When New Technology Emerges
- New Tools
- Tools - Compensating for Bugs
- Evolutions
- Chain of Events
- Common Fear
- Perspective

- Relevant Revolutions
- Performance Hits
- Improving Performance
- Each Revolution
- OOP and Complexity
- Payoff

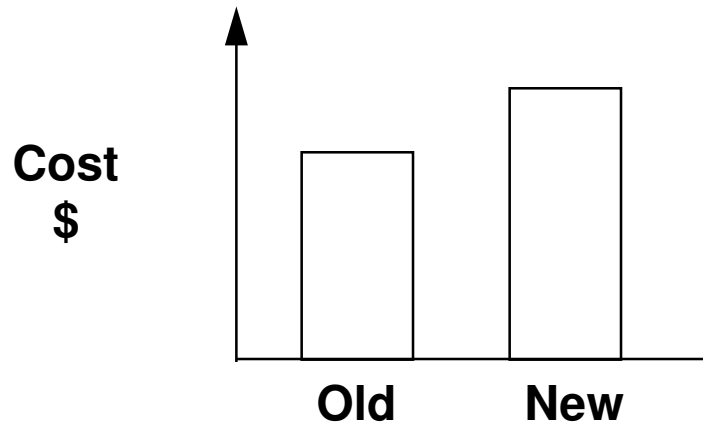
WHEN NEW TECHNOLOGY EMERGES

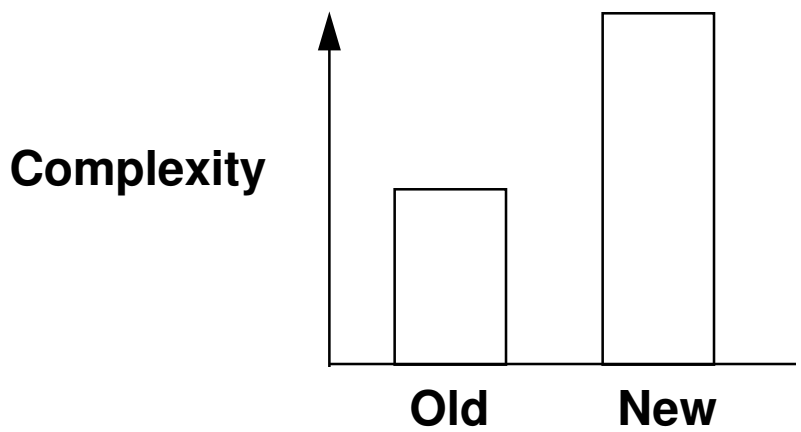
Hardware



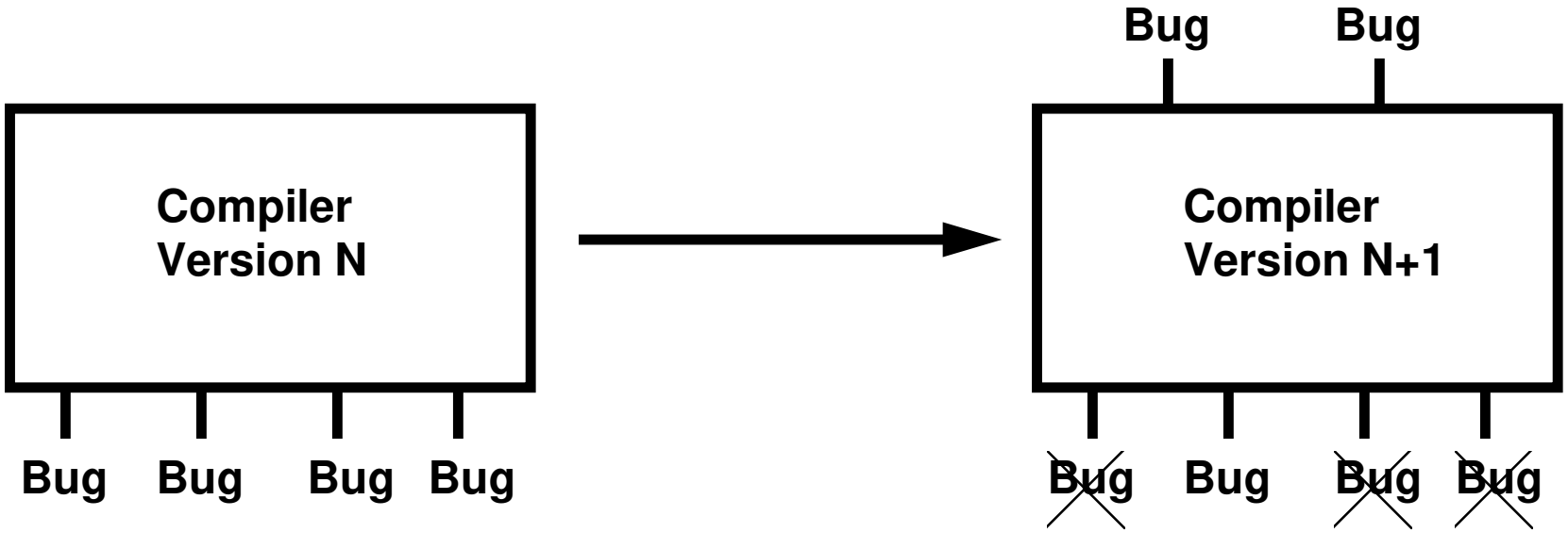


Software



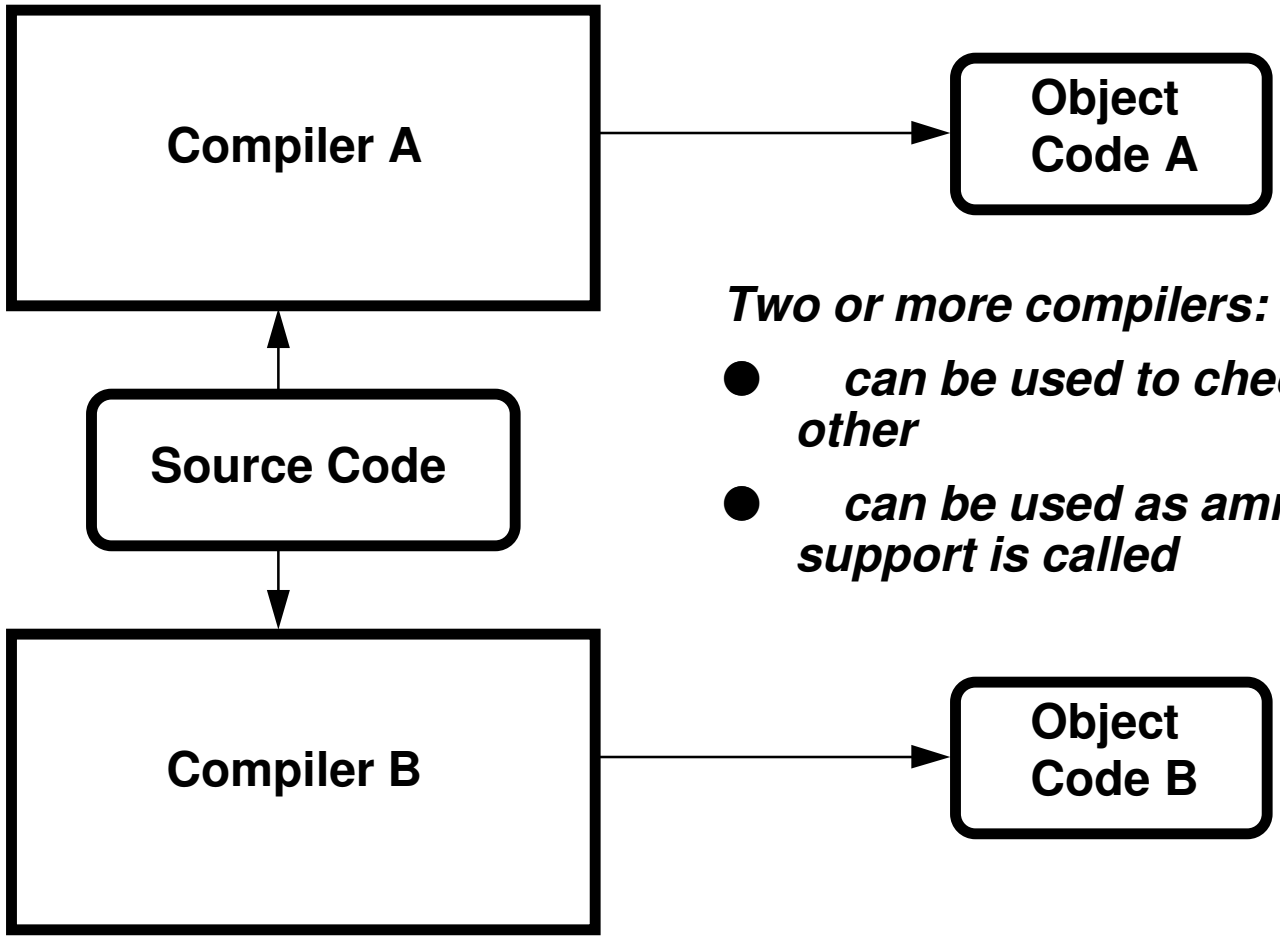


NEW TOOLS



Some Old Bugs Fixed → Some New Bugs Introduced

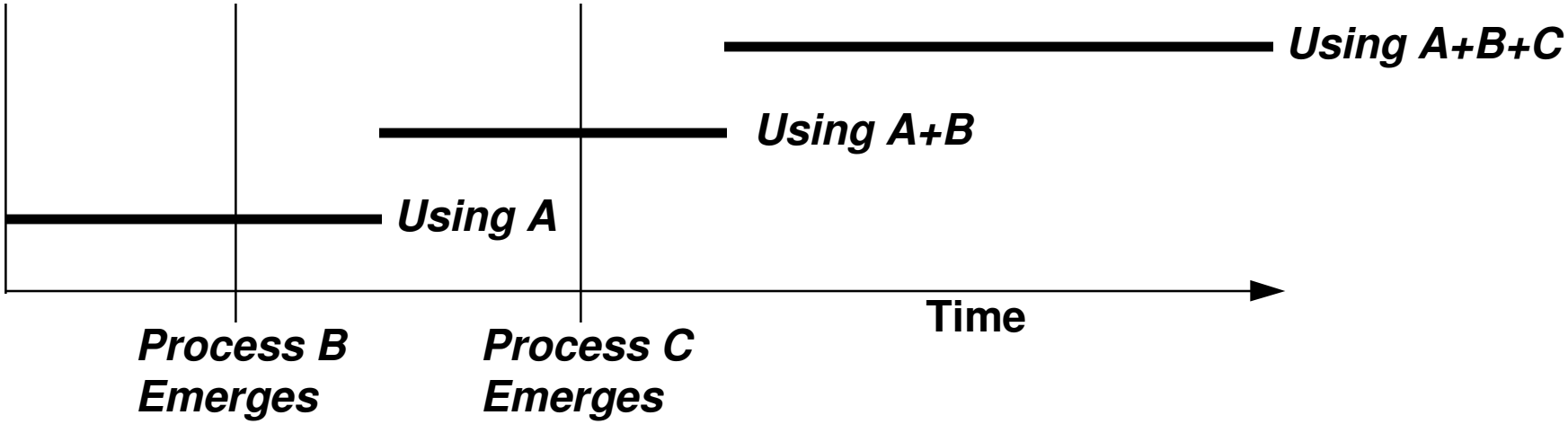
TOOLS - COMPENSATING FOR BUGS



Two or more compilers:

- ***can be used to check each other***
- ***can be used as ammo when support is called***

EVOLUTIONS



The promoters of new developments often start with hype:

- **They promise incredible productivity improvements.**
- **They predict the imminent demise of all older styles of programming.**
- **They hint that old-timers will be replaced by trained newcomers.**

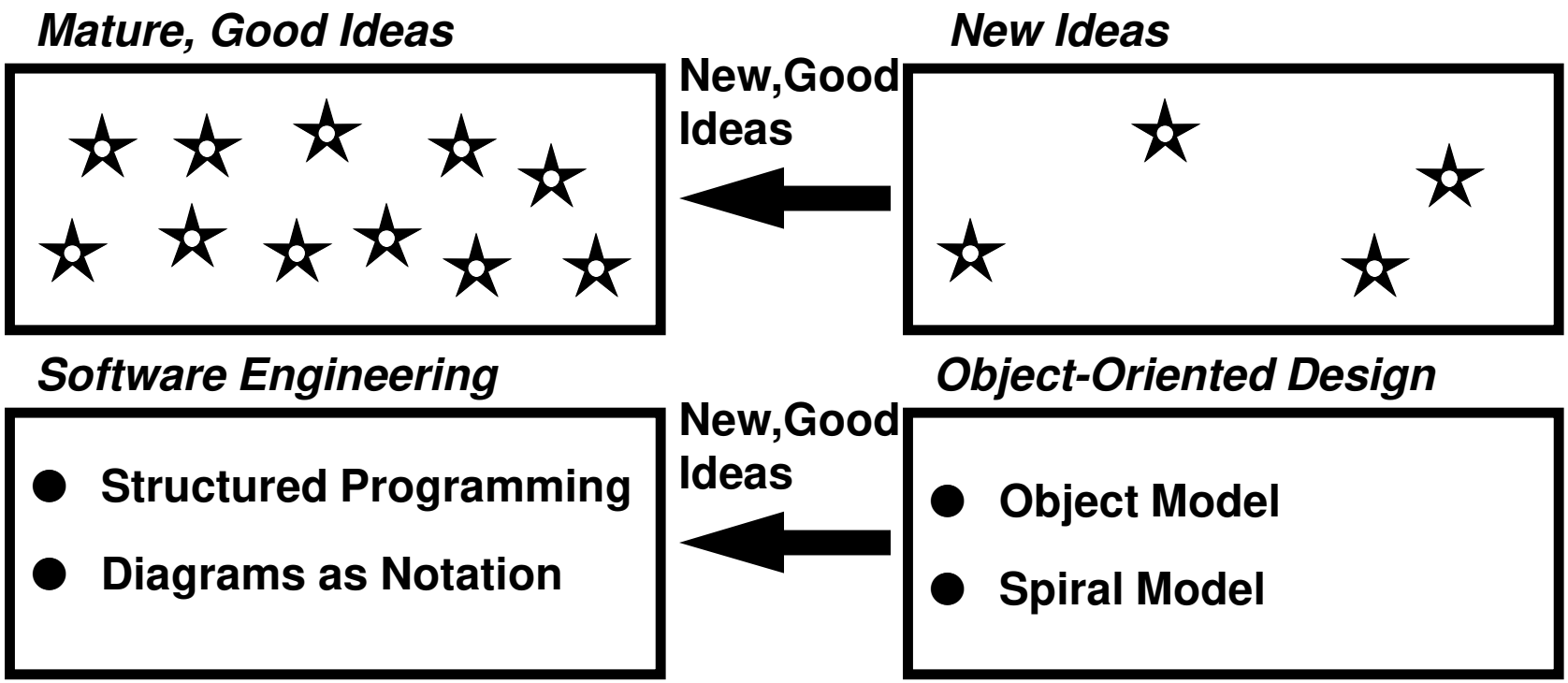
On the whole, the promoters are wrong!

The problem with being a zealot:

- **If you are right, the world will little note nor long remember it.**
- **If you are wrong, you will never live it down.**

CHAIN OF EVENTS

1. Zealots preach and practice new techniques while the practitioners wait and watch.
2. What the zealots did right became apparent eventually; what they did wrong also became apparent.
3. The practitioners kept the good bits and discarded the rest.



A COMMON FEAR

We will become obsolete.

Do not fear if:

- **you balance conservatism with flexibility**
- **you watch, wait, and know how long to wait**
- **you then try the best ideas first**

Keep the faith:

- **Realize what you are getting done with your current tools.**
- **Compare your productivity today with that of a few years ago.**
- **As a result of delivering significant software products with some reliability, you will always be asked to do more ambitious jobs.**

We are suffering from success, not failure.

LET'S PUT OBJECT-ORIENTED PROGRAMMING IN PERSPECTIVE!

RELEVANT REVOLUTIONS

- 1960-85: Higher level languages displace assembly languages.
- 1965-80: Structured programming improves the way we wright higher level languages.
- 1970-80: Modular programming replaces monolithic programming.
- 1975-80: Structured design improves the way we modularize programs.
- 1980-90: Structured analysis improves the way we identify and group modules.
- 1985-??: Object-oriented design and programming introduce a new way of grouping modules and data.

Note that the starting dates are every 5 years:

1. A few years are required to digest the approach.
2. Time is needed to rest and regroup, getting ready for the next new idea.
3. Earlier revolutions took longer to settle than the new ones - they reflect the accelerating pace of software technology and maturity.

PERFORMANCE HITS A COMMON EARLY PROBLEM

- Early compilers often produced code 2 to 10 times slower than assembly language. Today's compilers occasionally do *better* than hand-crafted code.
- Early structured programs were often larger and occasionally slower than unstructured programs. Better control statements in modern languages and better compilers have eliminated this concern.
- Modular programming introduces numerous function calls. These were expensive on many architectures of the 1960's. Modern architectures have all but eliminated the penalty of calling numerous functions.
- Structured design adds even more functions. On newer machines with virtual memory, however, it actually improves performance by improving locality of reference.
- Structured analysis narrows interfaces and, subsequently, increases argument passing. Again, improved architectures eliminate most of this penalty.

IMPROVING PERFORMANCE

ARGUMENT: Object-oriented languages cost too much in performance.

COMMON BELIEF: We pay for our software sophistication by eating more hardware. Rapid advances in hardware performance subsidize our taste for more complex programming methods.

This may be true in the short run, but not over the long haul. Early performance hits often occur because the hardware is *unsuitable* for the new way of doing business.

Adapt the architecture to the discipline, and performance improves.

The demand for faster and larger computers comes primarily from the applications, not the tools.

EACH REVOLUTION DOES *NOT* OBSOLETE THE PREVIOUS TECHNOLOGIES!

OOP makes you think of a program as a grouping of objects, but those objects perform their activities via functions. The newer technology is still supported by the older technology.

OOP has recently come into its own because of the complexity of our applications today:

- **We need an additional level of grouping, or abstraction, to handle more complex problems, and OOP gives us this abstraction.**
- **We need to restrict side effects and semantics, and OOP enforces this restriction.**
- **We need better controls on initialization and deinitialization of segments of a system, and OOP provides this control.**

OOP AND COMPLEXITY

OOP, then, meets some needs as a system becomes complex. But if the system is *not* complex, OOP can get in the way, placing an extra, perhaps unneeded, layer in the software.

As the programs grow,

1. **It becomes harder to remember what is to be done.**
2. **Subtle usage bugs get introduced.**
3. **Reliability starts to go down.**

We become willing to sacrifice some performance for safety and reliability:

1. **We want the compiler to tell us as soon as possible when we are doing something wrong.**
2. **We want certain problems to never get into the executable, being trapped by the compiler long before it gets to the executable.**

We want such enforcement with little or no cost in performance.

THE PAYOFF TO ADOPTING OOP IS WORTH THE EFFORT PROVIDED YOU DON'T INVEST TOO MUCH EFFORT.

The trick lies in introducing objects where they really pay off.

Treat OOP as the next discipline to apply when you are suffering from too much success using what you already know.