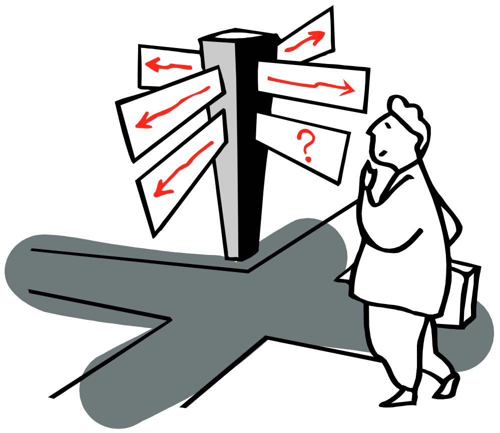
5. Testing and Migration

- What and Why
  - + Reengineering Life-Cycle
- Tests: Your Life Insurance !
  - + Grow Your Test Base Incrementally
  - + Use a Testing Framework
  - + Record Business Rules as Tests
  - + …
- Migration Strategies
  - + Make a Bridge to the New Town
- Conclusion



What and Why ?

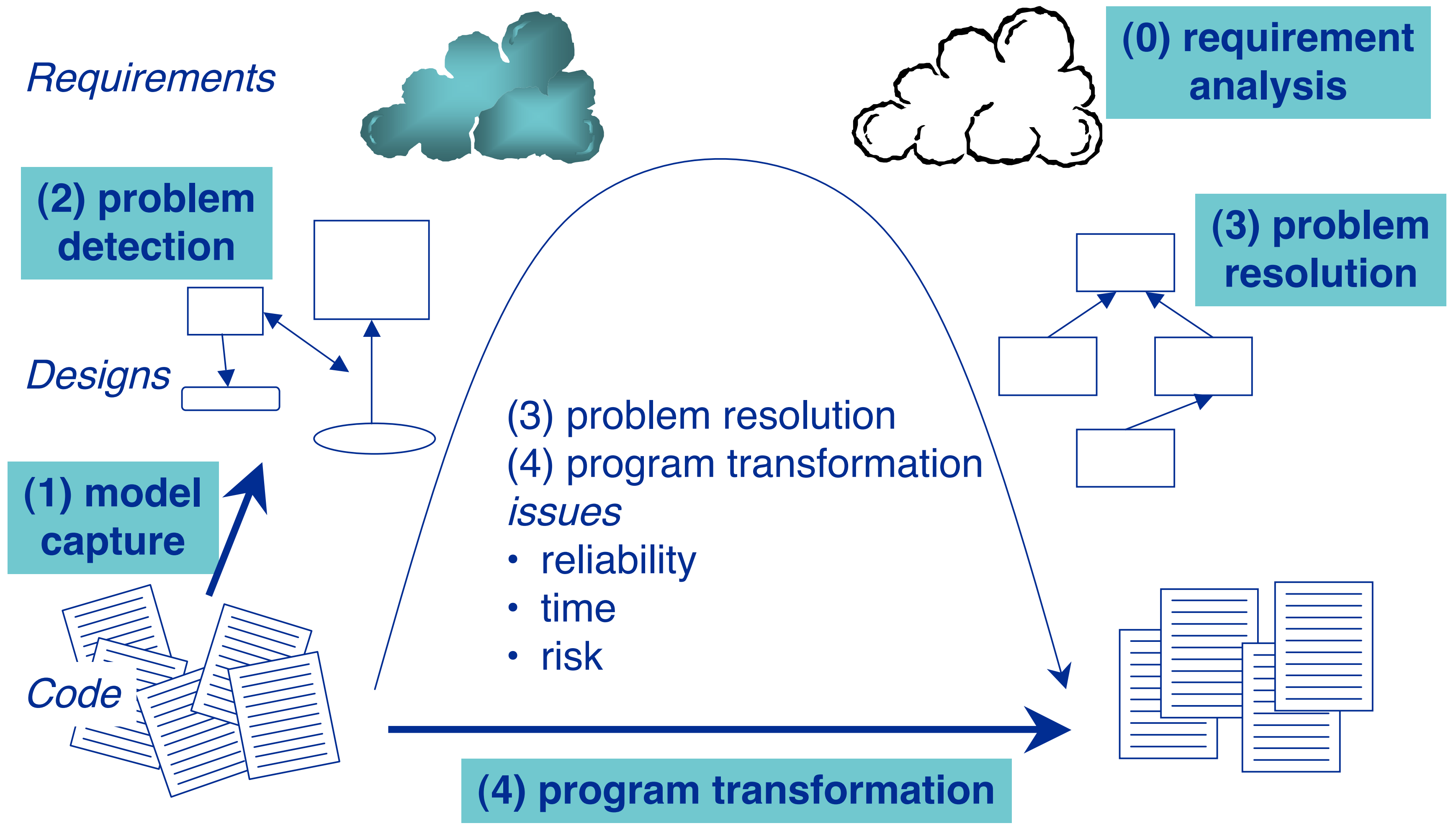
Definitions

- *Restructuring* refers to transforming a system from one representation to another while remaining at the same abstraction level. — Chikofsky & Cross, '90
- *Refactoring* is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure — Fowler, '99

Motivation

- Alter the source-code to
  + solve *problems* identified earlier
  + without introducing new *defects*
  + and while the system remains in *operation*

The Reengineering Life-Cycle

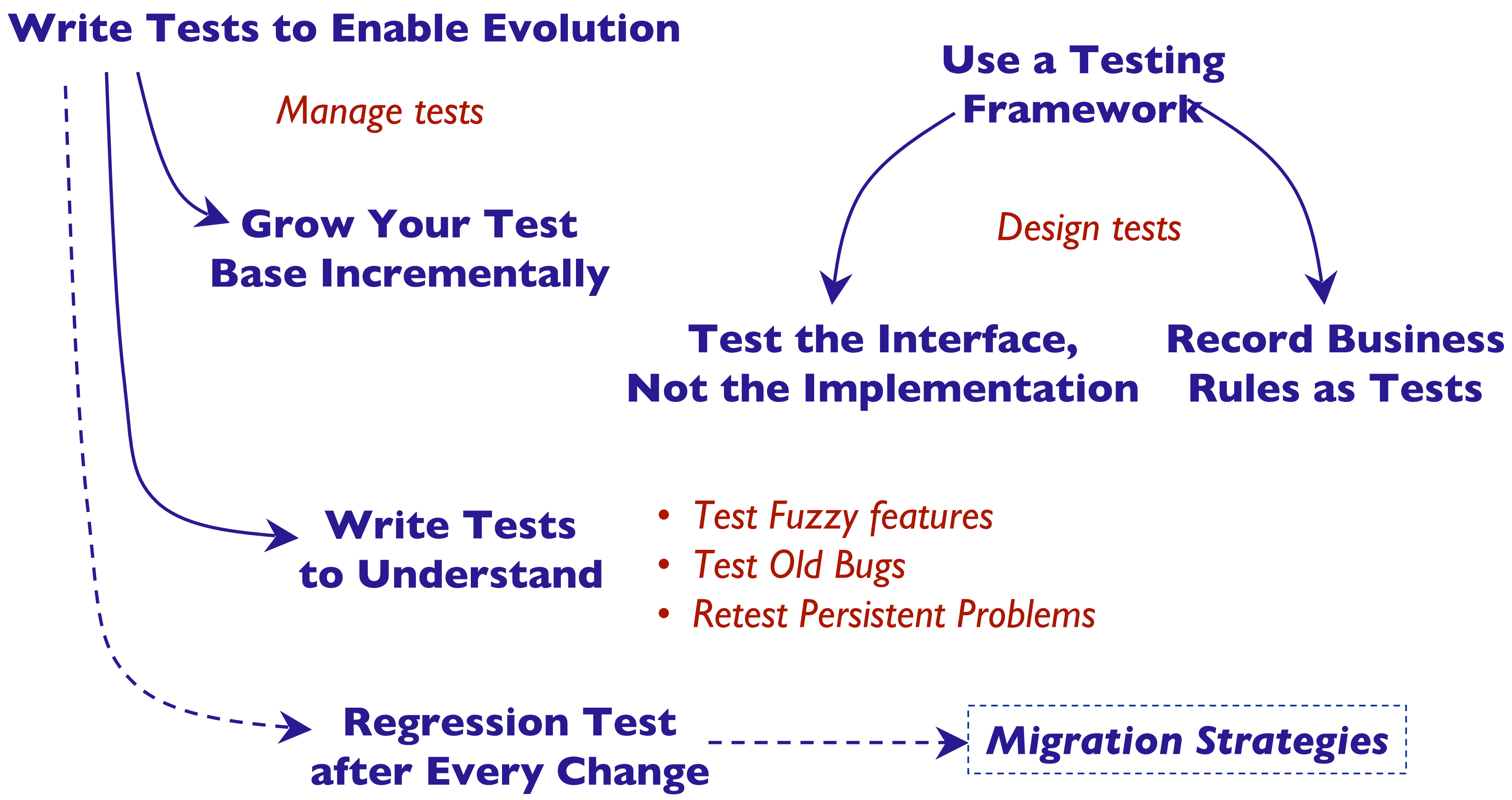


(3) problem resolution
(4) program transformation
issues
- reliability
- time
- risk

Forces — Testing

- Many legacy systems *don’t have tests*
- Software changes introduce *new bugs*
- You can’t test *everything*
- Concurrency and user interfaces are *hard to test*
- Testing is usually everyone’s *lowest priority*
- Knowledge *concentration* poses *high risk*
- Customers *pay for features*, not tests
- Customers don’t want *buggy* systems
- *Good* programmers *don’t need tests*
- New tools and techniques are more *fun* than testing
- Testing is akin to *street-cleaning*

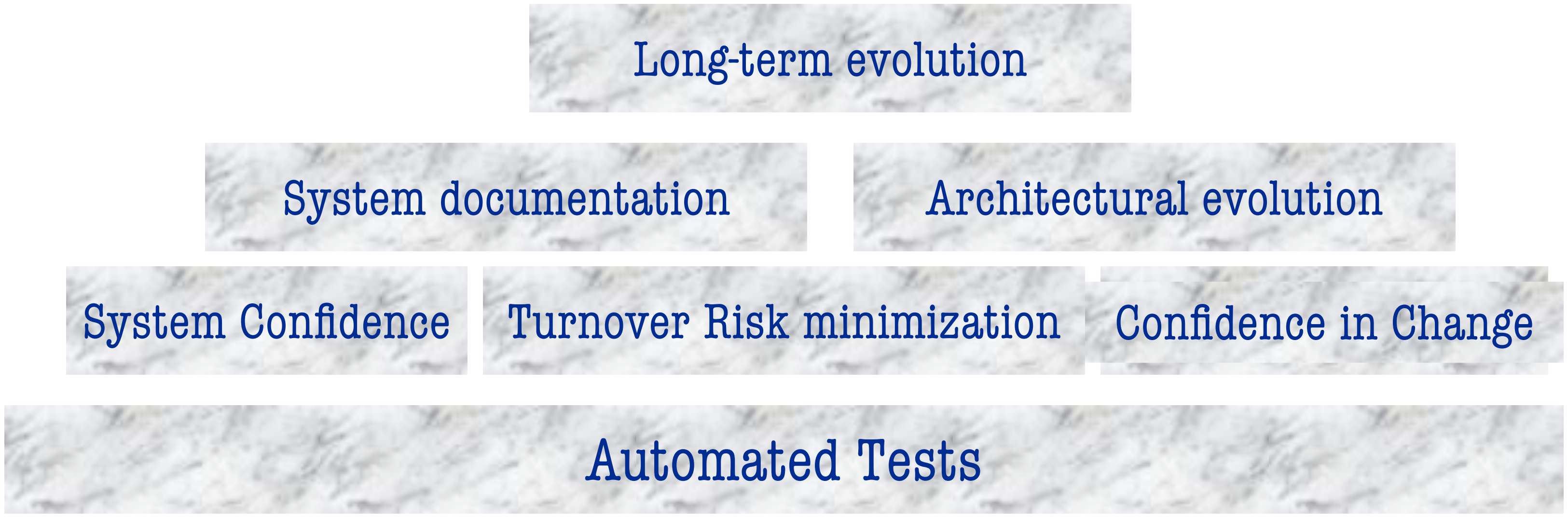
Tests: Your Life Insurance!


Write Tests to Enable Evolution
Manage tests
Grow Your Test Base Incrementally
Use a Testing Framework
Design tests
Test the Interface, Not the Implementation
Record Business Rules as Tests
Write Tests to Understand
• Test Fuzzy features
• Test Old Bugs
• Retest Persistent Problems
Regression Test after Every Change
Migration Strategies

Write Tests to Enable Evolution

Problem: How do you minimize the risks of change?

Solution: Introduce *automated, repeatable, stored* tests



Automated tests are the *foundation* of reengineering

Grow Your Test Base Incrementally

Problem: When can you stop writing tests?

Solution: When your tests cover all the code!

… however

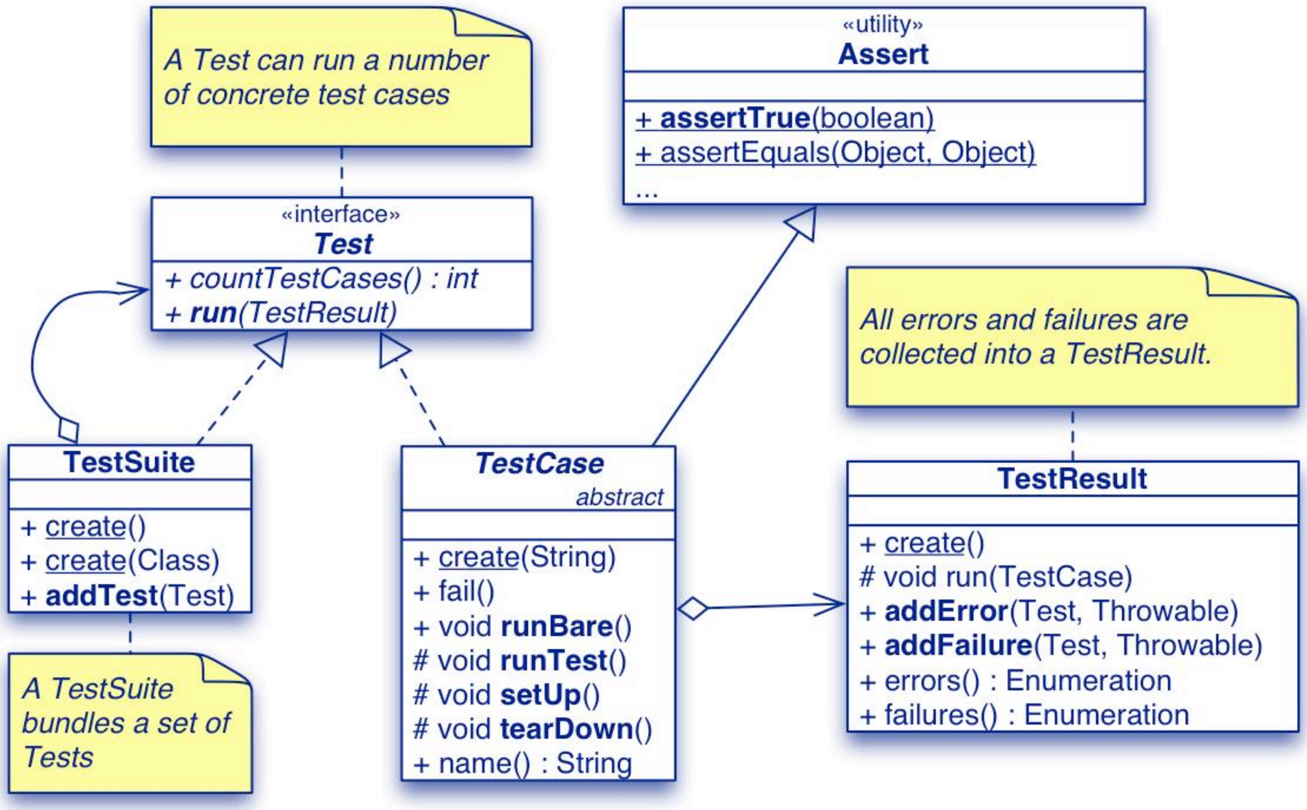
- + you're paid to reengineer, not to write tests
- + testing ALL the code is impossible
- + design documentation is out-of date
  - » *semi-automated black-box testing is not an option*

- Answer: Grow Your Test Base Incrementally
  - first test *critical* components (business value; likely to change; …)
  - keep a snapshot of old system (run new tests against old system)
  - focus on business values
  - test old bugs + new bugs that are reported

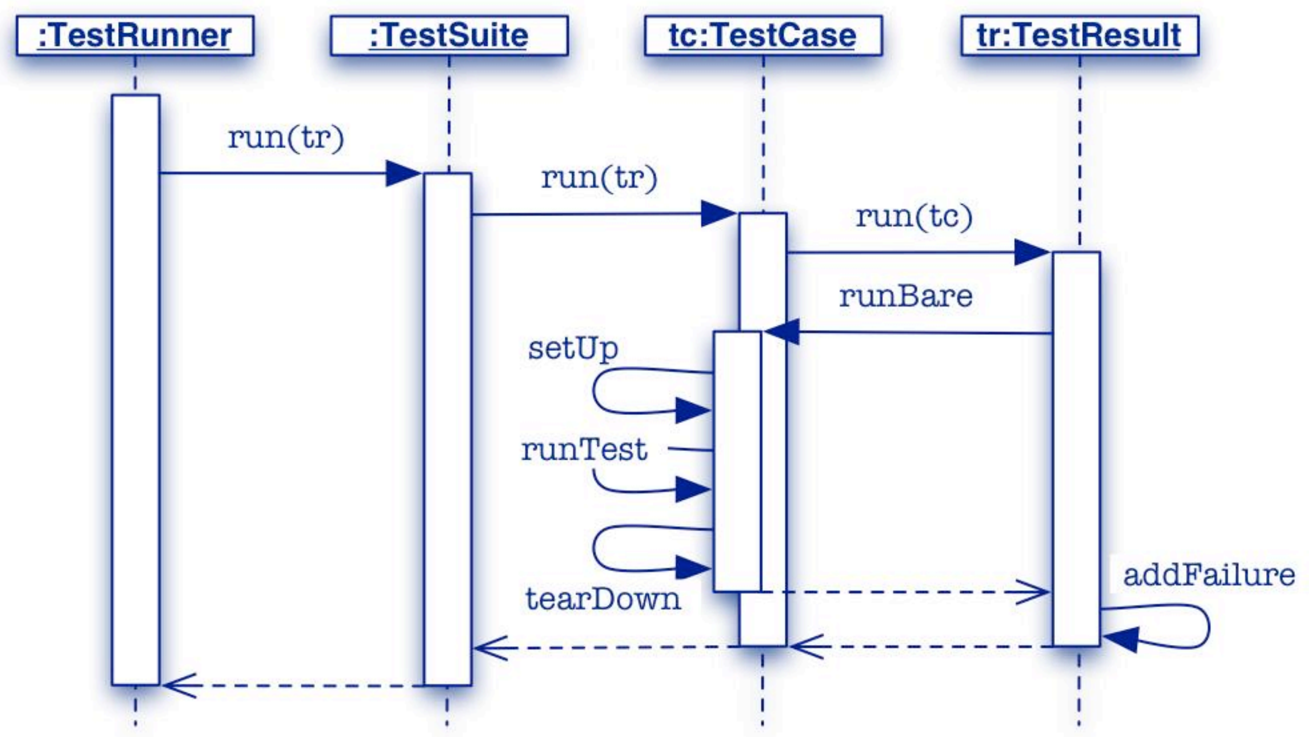
Use a Testing Framework

Problem: How do you encourage systematic testing?

Solution: Use a framework to structure your tests



Running tests



Write Tests to Understand

Problem: How to decipher code without adequate tests or documentation?

Solution: Encode your hypotheses as test cases

- *Exercise* the code
- Formalize your reverse-engineering *hypotheses*
- Develop tests as a *by-product*

Record Business Rules as Tests

Problem: How do you keep your system in sync with the business rules it implements?

A Solution: *Good documentation + Good design*

- … *however*
  - + business rules are too complex to design well
  - + documentation & design degrades when the rules change
  - + business rules become implicit in code and minds

Solution: *Record Business Rules as Tests*
- canonical examples exist
- can be turned into input/output tests

Example: Payroll Business Rule

A person or couple gets an amount of money for every child he, she or they raise. Basically parents get CHF 150,- per month for every child younger than 12 years, and CHF 180,- for every child between 12 and 18 and for every child between 18 and 25 as long as the child is not working and is still in the educational system. A single parent gets the full 100% of this money as long as he or she is working more than 50%. Couples get a percentage of the money that is equal to the summed working percentages of both partners.

Example: Payroll Test Case

```
"--- input-cases are extracted from a database"
singlePerson80WithOneKidOf5 := extract....
couplePerson40occupationWithOneKidOf5 := extract....
couplePerson100occupationWithOneKidOf5 := extract....
couplePersonWithOneKidOf14 := extract....

"--- tests compare expected output against actual output"
self assert: singlePerson80occupationWithOneKidOf5 moneyForKid
        = 150.
self assert: couplePerson40occupationWithOneKidOf5 moneyForKid
        = 150*4.
self assert: couplePerson100occupationWith2KidsOf5 moneyForKid
        = 150*2.
self assert: couplePersonWithOneKidOf14 moneyForKid
        = 180.
```

Other patterns

Retest Persistent Problems

+ Always tests these, even if you are making no changes to this part of the system

Test Fuzzy Features

+ Identify and write tests for ambiguous or ill-defined parts of the system

Test Old Bugs

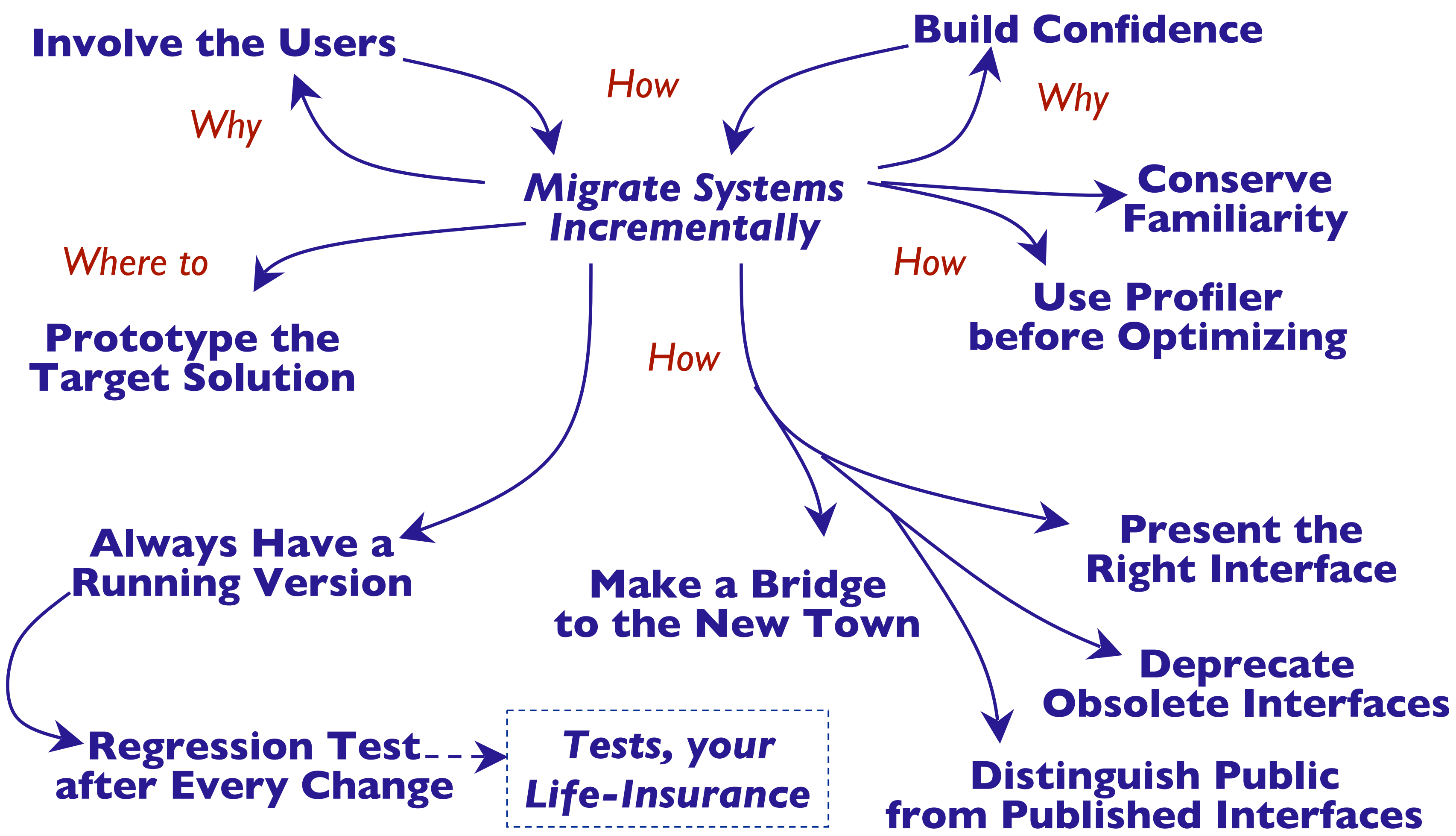
+ Examine old problems reports, especially since the last stable release

— *DeLano and Rising, 1998*

Forces — Migration

- Big-bang migration often *fails*
- Users *hate change*
- You need *constant feedback* to stay on track
- Users just want to *get their work done*
- The legacy data must be *available* during the transition

Migration Strategies


Involve the Users
Why
How
Build Confidence
Why
Migrate Systems Incrementally
Conserve Familiarity
How
Use Profiler before Optimizing
Where to
Prototype the Target Solution
How
Always Have a Running Version
Make a Bridge to the New Town
Present the Right Interface
Deprecate Obsolete Interfaces
Regression Test after Every Change
Tests, your Life-Insurance
Distinguish Public from Published Interfaces

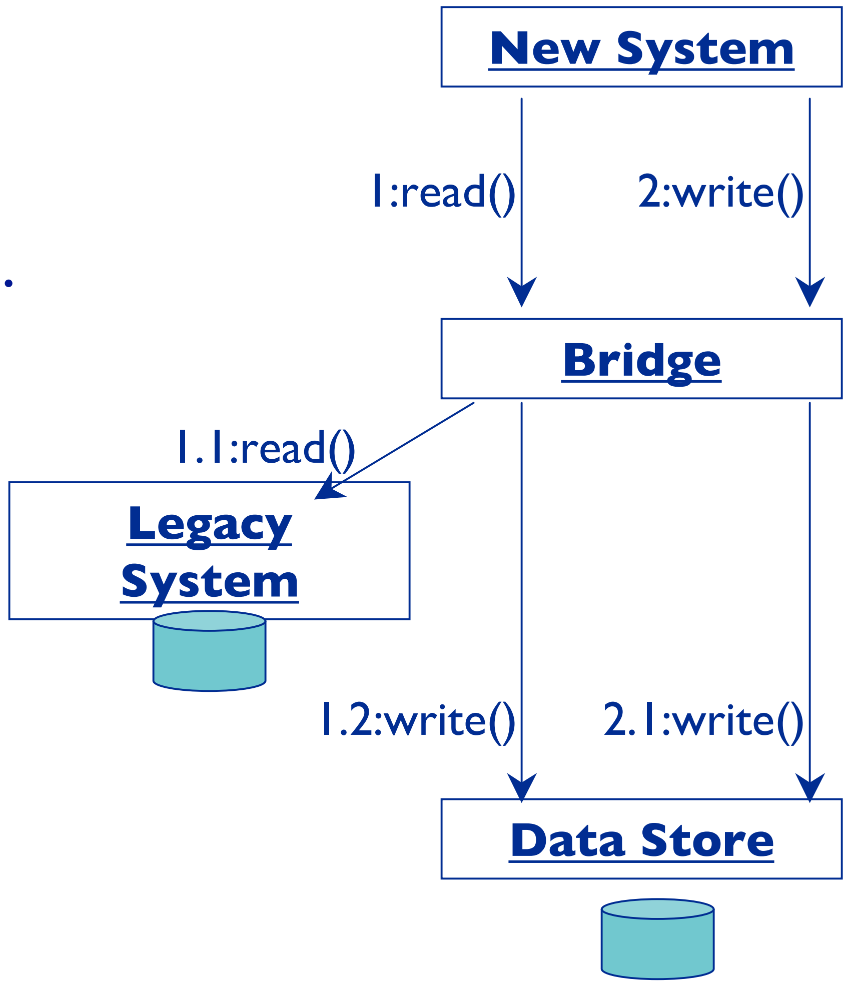
Make a Bridge to the New Town

Problem: How to migrate data?

Solution: Convert the underlying files/databases/…

... however

+ Legacy and new system must work in tandem
+ Too much data; too many unknown dependencies
+ Data is manipulated by components



Conclusion

Avoid risk

- + small increments ("chicken little")
- + develop suite of regression tests

… at acceptable cost

- + Migration costs as much as new development !
- + But you avoid "hidden costs"
  - team morale in maintenance team
  - satisfying two customer bases