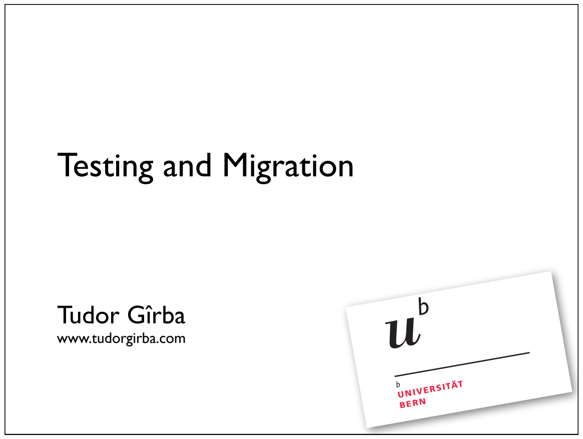
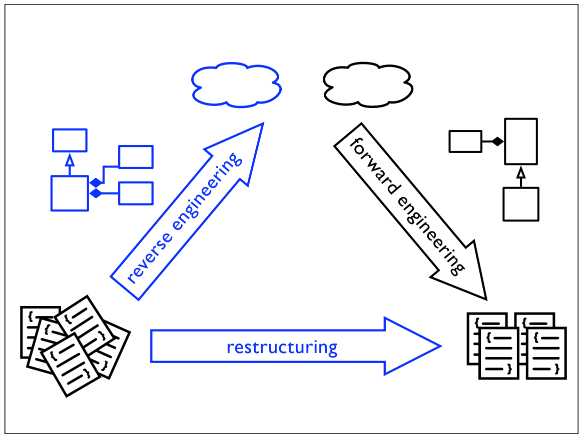


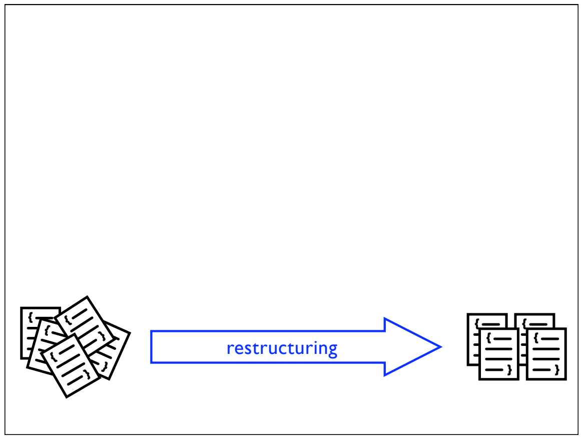
Testing and Migration

Tudor Gîrba
www.tudorgirba.com

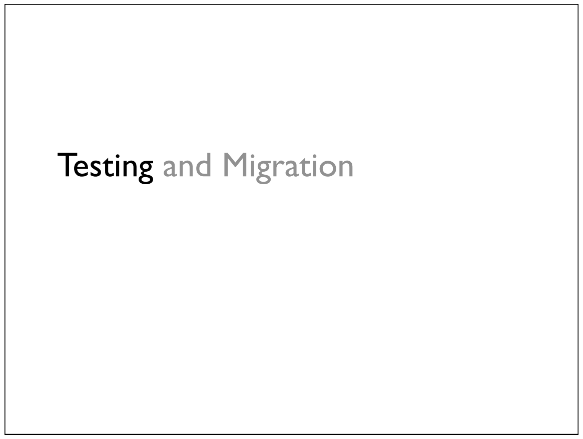


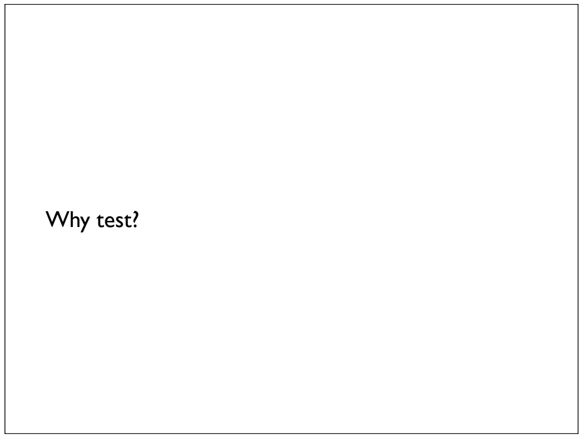
Reengineering ... is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

Elliot Chikofsky and James Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, January 1990, pp. 13—17.

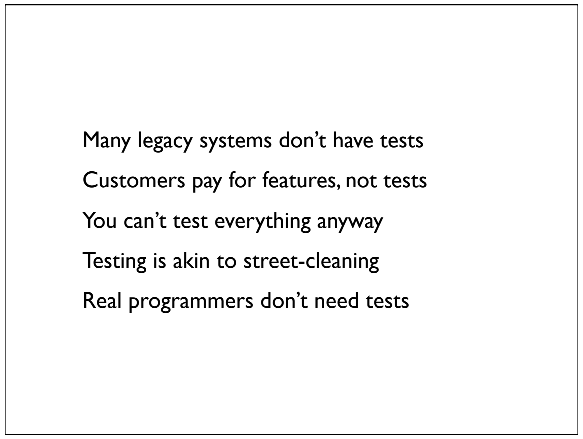


Restructuring is transforming a program to fit current needs.

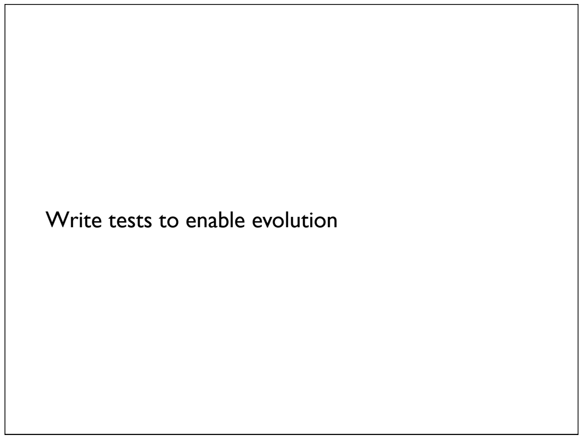

Testing and Migration


Why test?

Why should we test?

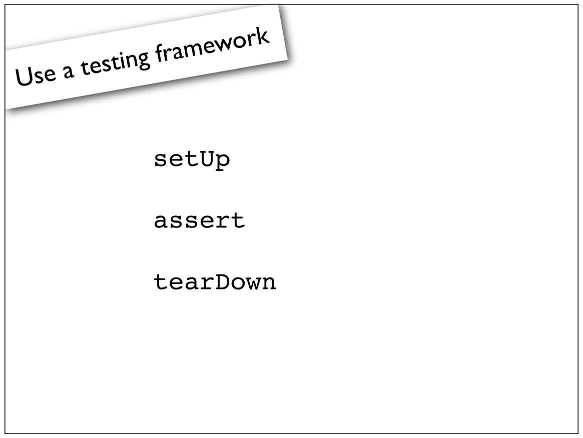

- Many legacy systems don’t have tests
- Customers pay for features, not tests
- You can’t test everything anyway
- Testing is akin to street-cleaning
- Real programmers don’t need tests

Testing is usually everyone’s lowest priority, but:
- Changes introduce new bugs
- Customers don’t want buggy systems



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



Use a testing framework

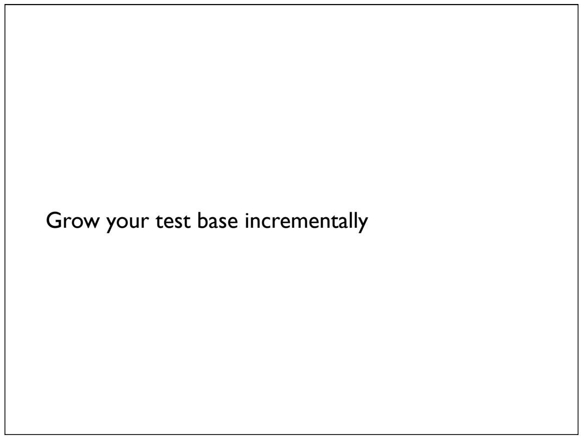
```
setUp

assert

tearDown
```

Problem: How do you encourage systematic testing?
Solution: Use a framework to structure your tests

Does it seem old news for you?
Hopefully yes :). But, you would be surprised to see how many projects don't do it.



Grow your test base incrementally

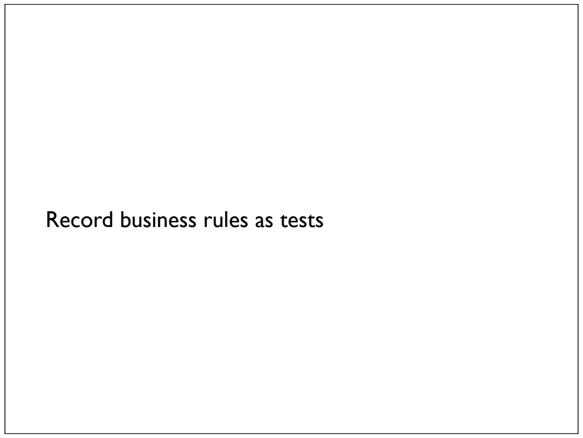
Problem: When can you stop writing tests?
Solution: When your tests cover all the code! :)
... but
- you're paid to reengineer, not to write tests
- testing ALL the code is impossible
- design documentation is out-of date

Grow Your Test Base Incrementally
- first test critical components (business value; likely to change; …)
- focus on business values (test old bugs + new bugs that are reported)

Test the interface, not the implementation

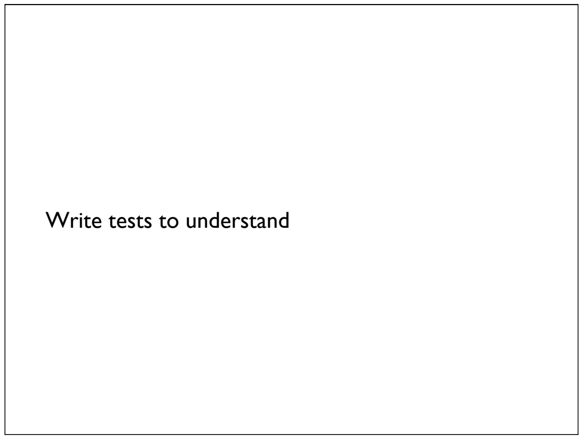
Problem: How do you protect your investment in tests?
Solution: Apply black-box testing

Test interfaces, not implementations. Be sure to exercise the boundaries
Test scenarios, not paths. Use tools to check for coverage
Beware: Enabling testing will influence your design!



Record business rules as tests

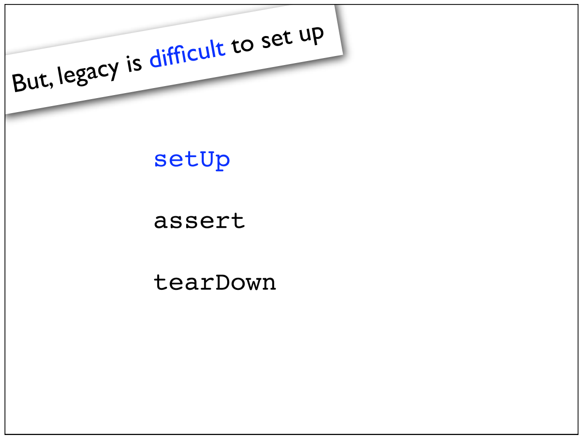
Problem: How do you keep your system in sync with the business rules it implements?
One solution: Good documentation + Good design … however
- Business rules are too complex to design well
- Documentation & design degrades when the rules change
- Business rules become implicit in code and minds
Solution: Record Business Rules as Tests
- canonical examples exist
- can be turned into input/output tests



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

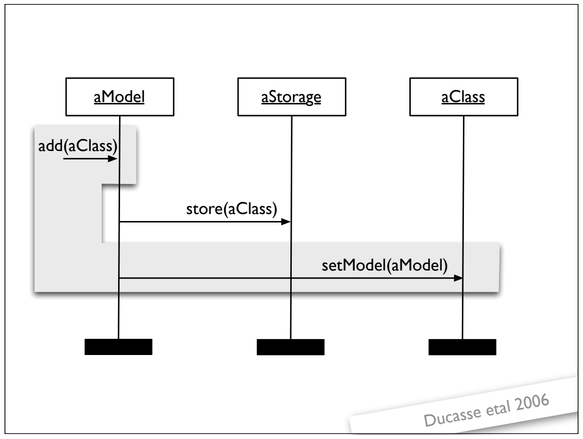

But, legacy is difficult to set up

```
setUp

assert

tearDown
```

To test, we first need to bring the system into a desirable state. However, due to the complexity of the system, poor design and lack of documentation, it is very difficult to write a setup for the test.



One solution is to assert conditions on a trace. For this, we just need to instrument the system, execute desired functionality from the user interface, and then write the assertion on the captured trace.

This picture shows a compressed view an execution trace.

Stéphane Ducasse and Tudor Gîrba and Roel Wuyts. Object-Oriented Legacy System Trace-based Logic Testing. In Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06), p. 35--44, IEEE Computer Society Press, 2006.

Testing and Migration

Migration is a restructuring that
changes the underlying infrastructure

1989

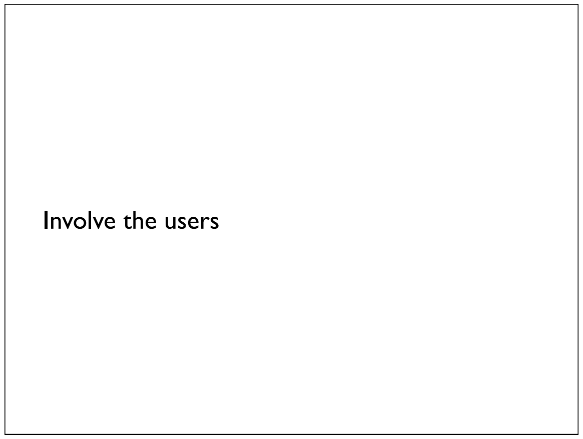
Big-bang migration often fails

Users hate change

You need constant feedback to stay on track

Users just want to get their work done

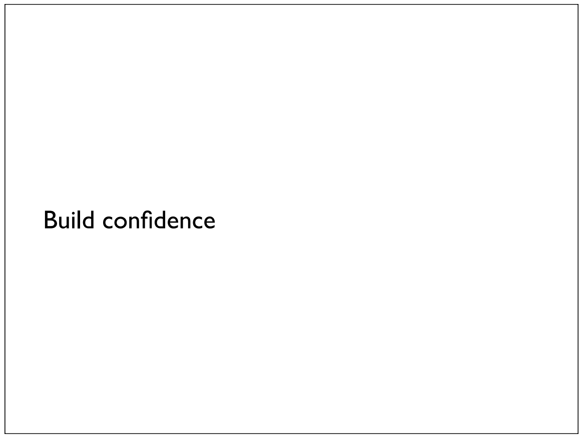
The legacy data must be available during the transition



Involve the users

Problem: How to get users to accept change?
Solution: Get them involved by giving them what they want

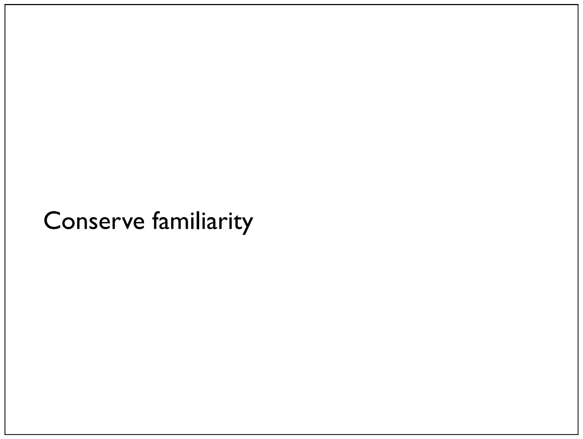
Start with the Most Valuable First.
Prototypes can help raise enthusiasm, but may also raise expectations too high.
Deploy early to increase commitment
- Diverts energy from development
- Pays back in quality feedback



Build confidence

Problem: How do you overcome skepticism?
Solution: Deliver results in short, regular intervals

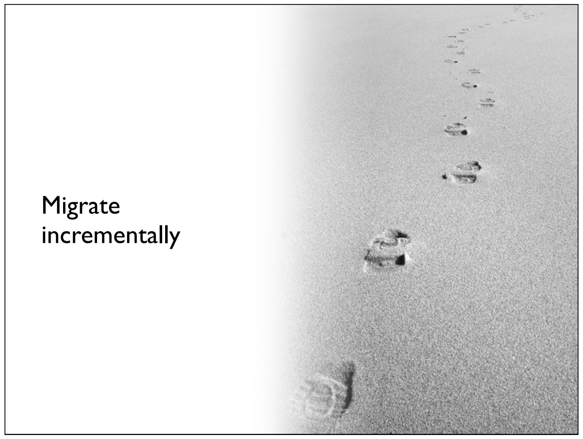
Requires time to sync with users.
Requires effort to support the changes.
Requires care not to alienate original developers.
Requires regular, big demos to convince management.



Conserve familiarity

Problem: How to avoid disrupting Users' work?
Solution: Avoid radical changes

Avoid alienating users
Introduce a constant, small number of changes with each release



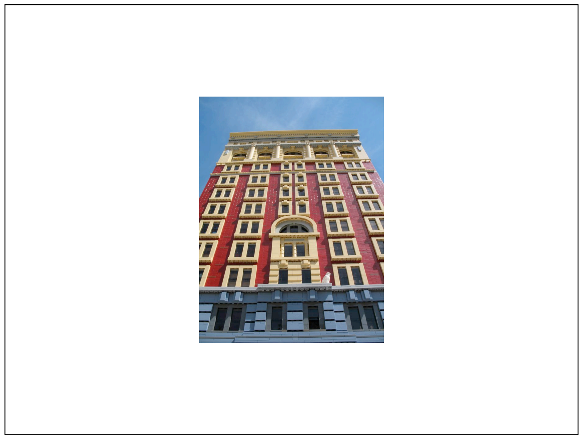
Migrate incrementally

Problem: When should you deploy the new system?
Solution: As soon as possible
Decompose the legacy system into parts
Tackle one part at a time (Most Valuable First)
Put suitable tests in place
Decide whether to wrap, reengineer, or replace
Deploy, support and obtain feedback
Iterate

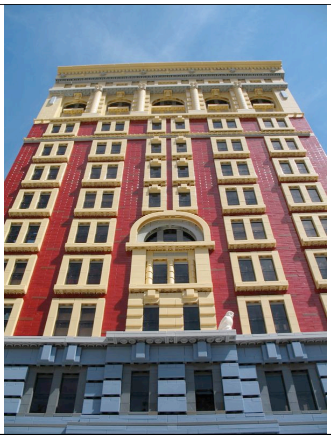
Prototype the target solution

Problem: How do you evaluate the target solution?
Solution: Develop a prototype

Evaluate the technical risks
- New system architecture
- Migrating legacy data
- Adequate performance …



Would you like to live in this house?



How about in this one?

Throw-away vs. evolutionary prototypes

Prototypes are of two types: throw-away and evolutionary. Evolutionary prototypes are build with the long-term intention of building on them the real system. Throw-away prototypes, on the other hand, are built to assess risks or test ideas.

However, from outside, a throw-away prototype can look like an evolutionary one and can be mistaken for the real system. You have to state from the beginning if a prototype is a throw-away one and defend the point of view against shallow points of view.

Always have a running version

Problem: Maintaining confidence during development
Solution: Integrate changes on a daily basis

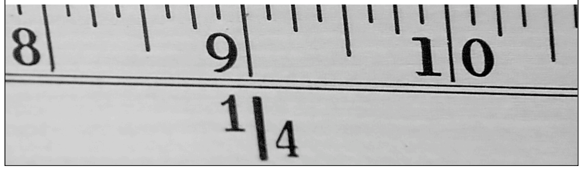
Use version and configuration management tools
Maintain exhaustive regression tests where needed
Plan short iterations — Continuous Integration
If necessary, re-architect the system to enable short build times

Test after every change

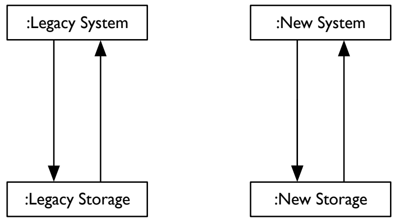
Problem: Making sure changes don’t break the system
Solution: Run the regression tests at each “stable” point

You must relentlessly write tests!
Write new tests whenever new (untested) bugs are discovered.
Take time to convince your team of the Joy of Testing.
If testing takes too long, categorize tests.
But run all the tests at least once a day.
Consider writing tests up front.
Remember to Retest Persistent Problems.

Use the profiler before optimizing



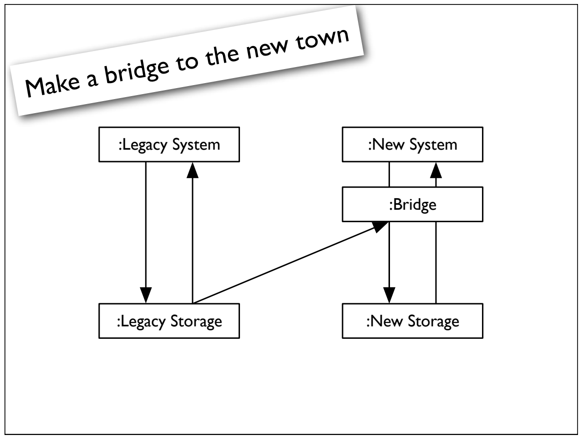
Do it, do it right, do it fast.



$$\begin{tikzcd}
\text{:Legacy System} \arrow[d, shift right] & \text{:New System} \arrow[d, shift right] \\
\text{:Legacy Storage} \arrow[u, shift right] & \text{:New Storage} \arrow[u, shift right]
\end{tikzcd}$$

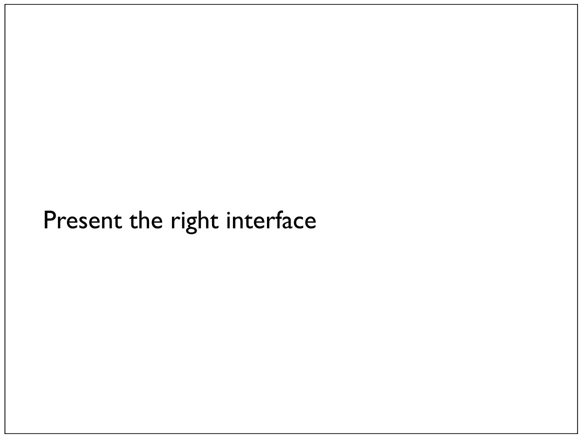
Data needs to be preserved and available.

However, migrating legacy storage to a new storage is not always trivial.



Make a bridge to the new town

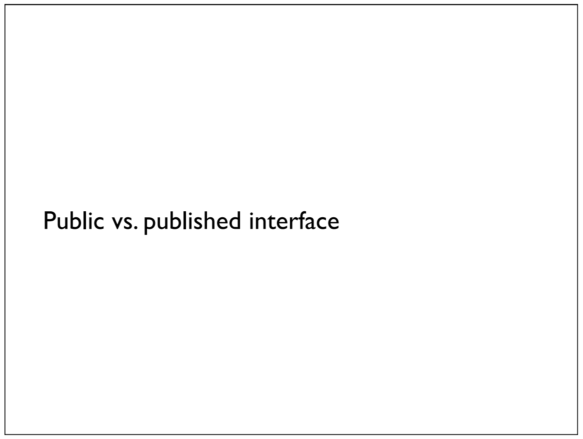
Problem: How to migrate data?
Solution: Convert the underlying files/databases/…
... however
Legacy and new system must work in tandem.
Too much data; too many unknown dependencies.
Data is manipulated by components.



Present the right interface

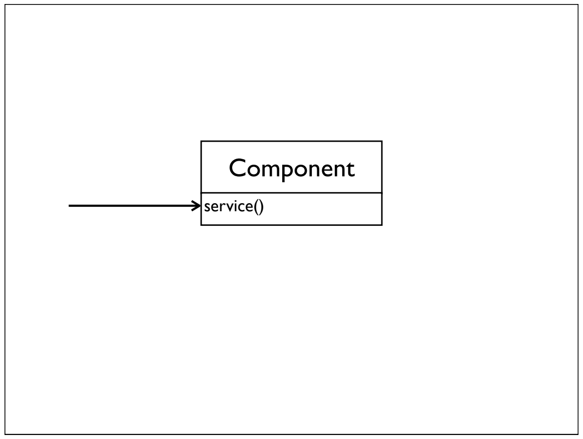
Problem: How do you prevent the legacy design from polluting the new system?
Solution: Wrap old services as new abstractions

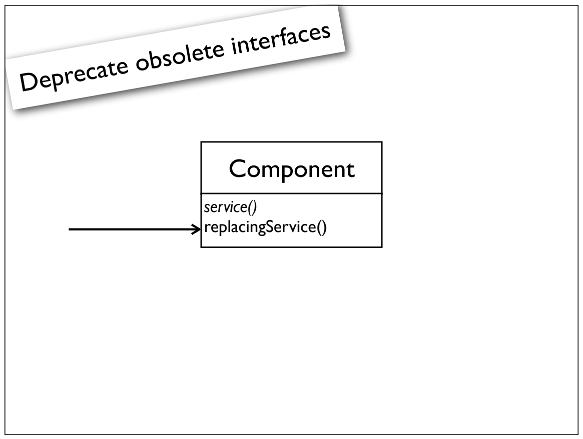
Identify the new abstractions you want.
Wrap the legacy services to emulate the new interface.
Avoid directly accessing old procedural interfaces.
Avoid wrapping as pseudo-OO «utility» classes.



Public vs. published interface

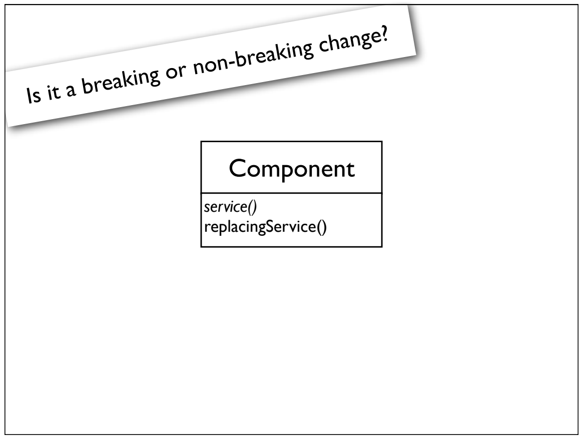
Problem: How to design interface for target solution?
Solution?: Think deeply ... however:
- Enable migration to target system ASAP.
- Avoid freezing the interface of target component.
- Costly ripple-effects of changes to public interface.
Solution: Distinguish between “public” and “published” interface
- public = stable target interface
- published = available, but unstable (use at your own risk)
Language features (protected, friends, …), naming conventions





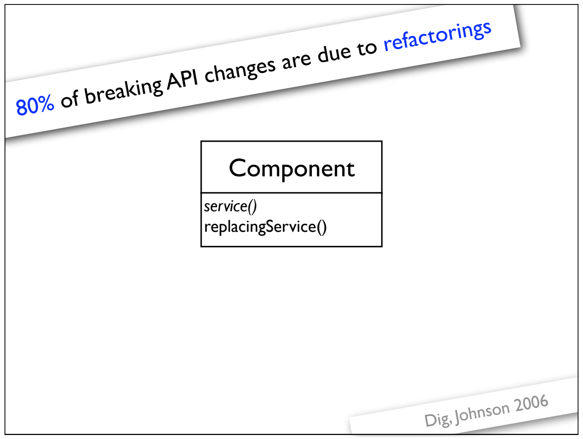
Problem: How to modify an interface without invalidating all clients?
Solution: Flag the old interface as «deprecated»

Old and new interfaces can co-exist for a time.
Deprecated usage can be lazily patched.
Various techniques possible
- Documentation (easy to ignore)
- Move or rename old interfaces (painful)
- Add warnings to deprecated code (should be non-intrusive)



Danny Dig and Ralph Johnson, How do APIs evolve? A story of refactoring, in Journal of Software Maintenance and Evolution (JSME), Volume 18, Issue 2, March/April 2006

An API breaking change is one that breaks the code of the clients. Code can either break at compile time, or at runtime because contracts are not preserved.


80% of breaking API changes are due to refactorings

Component

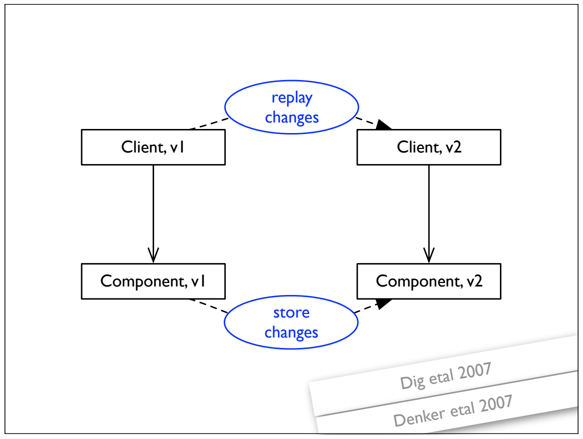
service()

replacingService()

Dig, Johnson 2006

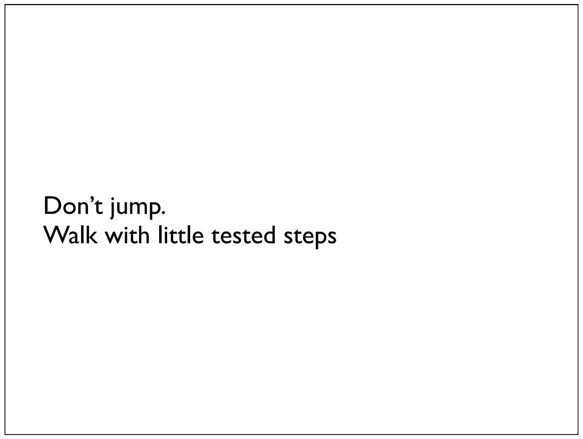
Danny Dig and Ralph Johnson, How do APIs evolve? A story of refactoring, in Journal of Software Maintenance and Evolution (JSME), Volume 18, Issue 2, March/April 2006

From a recent study, in 80% of the breaking changes from the Eclipse API were due to refactorings. If we would know which are these refactorings, we could re-apply them on the code of the client and dramatically reduce the cost of migrating from one version of the API to another.



Danny Dig, Kashif Manzoor, Ralph Johnson, Tien Nguyen: Refactoring-aware Configuration Management for Object-Oriented Programs, In Proceedings of ICSE'07, Minneapolis - May 2007

Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli and Pascal Zumkehr, “Encapsulating and Exploiting Change with Changeboxes,” Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), ACM Digital Library, 2007, pp. 25—49.


Don’t jump.
Walk with little tested steps

Tudor Gîrba
www.tudorgirba.com