Microsoft Research
FacultySummit

Beyond Behavior-Preservation

Ralph E Johnson
rjohnson@illinois.edu
University of Illinois at Urbana-Champaign

FUTURE/WORLD 2011 2031

Some of my PhD students

- Bill Opdyke – Refactoring Object-Oriented Frameworks, 1992
- Don Roberts – Practical Analysis for Refactoring, 1999 (The Smalltalk Refactoring Browser)
- Alejandra Garrido – Program Refactoring in the Presence of Preprocessor Directives, 2002
- Danny Dig – Automated Upgrading of Component-based Applications, 2007
- Munawar Hafiz – Security on Demand, 2010
- Jeff Overbey – A Toolkit for Constructing Refactorings, 2011 (Photran)

Importance of Refactoring

- Emphasize how programs change
- Step by step, keeping program running
- Incremental development -- Evolution, not revolution
- Integration with testing
- Automation, but programmer is in charge

Behavior-Preservation

- Refactorings change structure of program, not behavior
- Programs keep running, old tests still work, no new bugs
- Refactoring tools supposed to preserve behavior

- Reality
  - No tool can guarantee behavior-preservation
  - Programmers want to change behavior

Breaking Behavior-Preservation

- Rename not safe in reflective languages
- Not usually a problem to code owners
- If you don't know the code well:
  - Be conservative - warn of use of reflection
  - Detect common uses, such as names in XML

- Many changes almost behavior preserving
  - Parallelism – make faster, restrict how it is called
  - Security – preserve wanted behavior, prevent breakins
  - Reliability – preserve behavior when machines are perfect, recover when machines fail
  - Change from one DBMS to another
  - Add a new UI

Refactoring for parallelism

- Danny Dig – Concurrencer, Relooper, Immutator
- Introducing parallelism – Fork-join for divide-and-conquer, parallel arrays
- Making parallelism safer – atomic integers, lock-free maps, immutable objects

Refactoring for parallelism

- Similar to how programmers make programs more parallel
- Neither safe nor complete, but faster and more accurate than programmers

- Incremental
- Assist programmers,

Making immutable objects

- Value Objects – objects whose value is important, not their identity
  - Money, date, color, …
- Only assign instance variables in constructor
- Replace methods that modify instance variables with methods that return a new object
- Replace uses of object that need side-effects with a “holder”

Refactoring for security

- Munawar Hafiz – “Security on Demand”
- Describes 37 program transformations
- Eliminate buffer overrun by replacing unsafe string library with safe string library
- Eliminate injection attack by cleaning data
- Partitioning (Compartmentalization)
- Add access control

Security transformations

- Change behavior when system is attacked
- Preserve behavior when system is used as expected

Library Replacement

- char *strcpy (char *dst, const char *src) => gsize g_strlcpy (gchar *dst, const gchar *src, gsize dst_size)
- char *strcat (char *dst, const char *src) => gsize g_strlcat (gchar *dst, const gchar *src, gsize dst_size)

Partitioning

- Partition system so that breaking into one partition does not give access to other partitions
- Partitions should be separate address spaces, protected by operating system
  - Must replace pointers with other IDs

Library Replacement in general

- Optimization – general-purpose library => special purpose
- Reliability – use library that detects and recovers from failure
- Software evolution – obsolete => modern

Partitioning

- Reasons to change module boundaries
  - To promote reuse
  - To make system more secure
  - To make system easier to maintain
  - To make system more fault-tolerant
    - One part of the system crashes, other part can recover

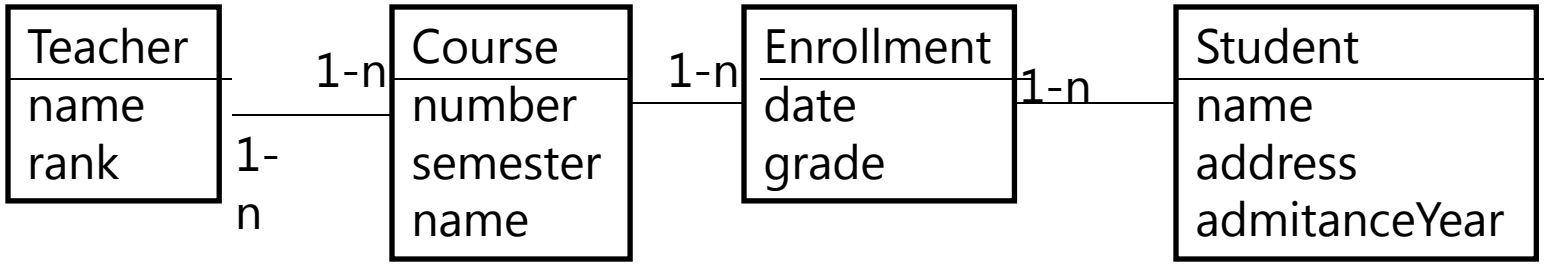
Enhancement

- Adding a feature requires changing API
  - Authorization requires knowing the user
  - Adding history to model requires knowing the effective date
- Changes
  - Internal data structures
  - Interface to module (API)
  - Tests

Example to enhance

- Teachers
  - Teach courses
  - Give grades to students in their courses
- Students
  - Take courses
- Courses
  - Students enroll in a course, and eventually get a grade

Example to enhance

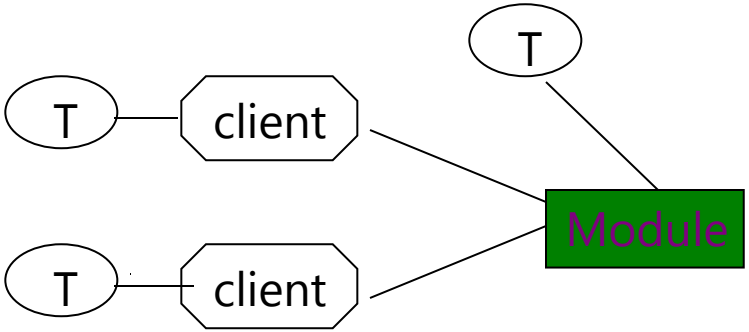


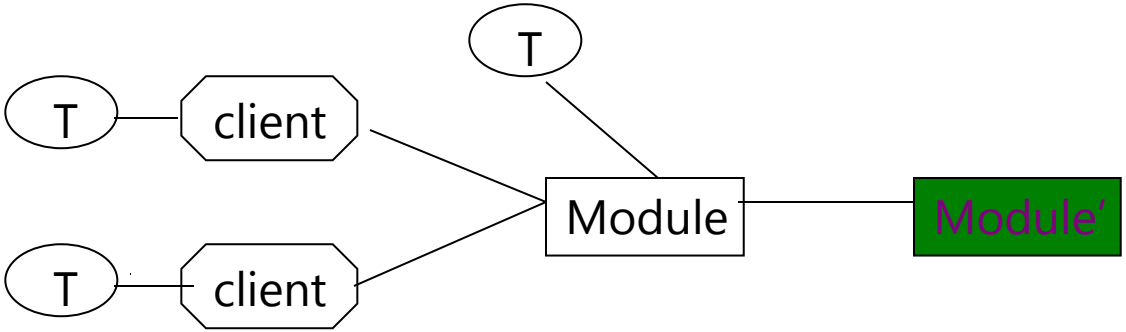
Enhancement: Role based access control

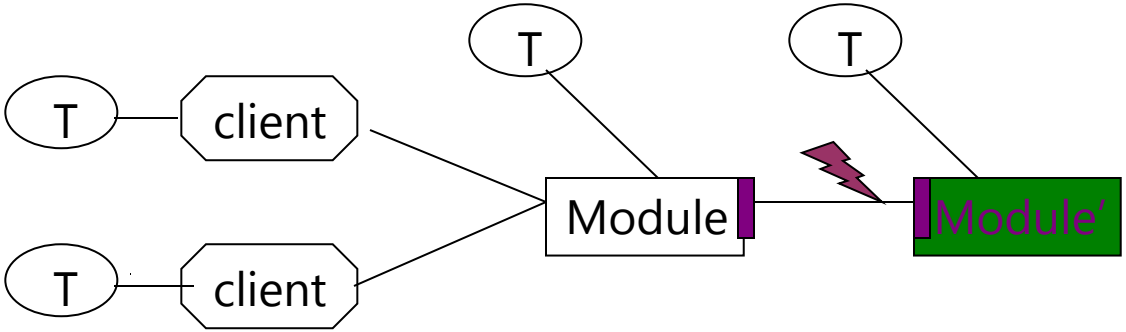
- Return type of all methods is changed; each method can fail
- Current user becomes part of API
  - Choice 1 - set “current user” before using API
  - Choice 2 - make “current user” be a parameter of all methods

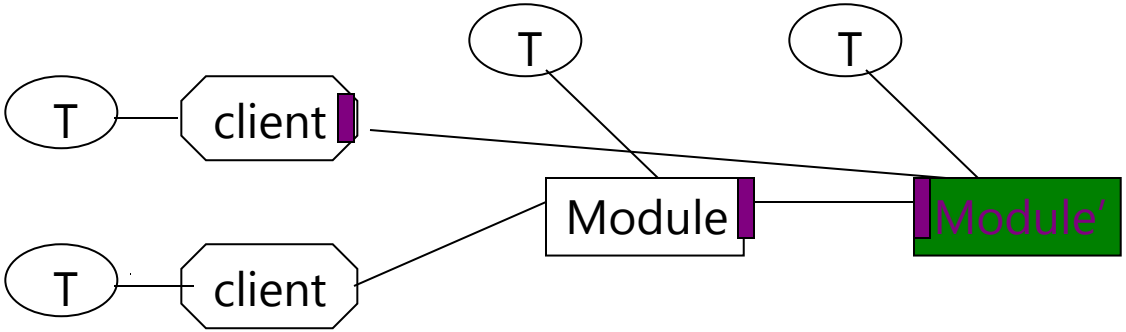
Enhancement: history

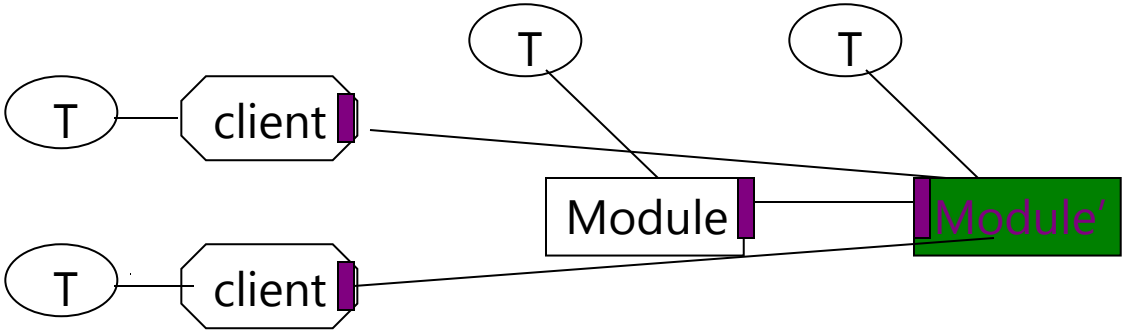
- Keep complete history of all name changes, creation of courses, and assignment of grades
- Change value at a particular time
- Read value at a particular time
- Time becomes part of API
  - Choice 1 - make “time” be global variable
  - Choice 2 - make “time” be parameter of each method

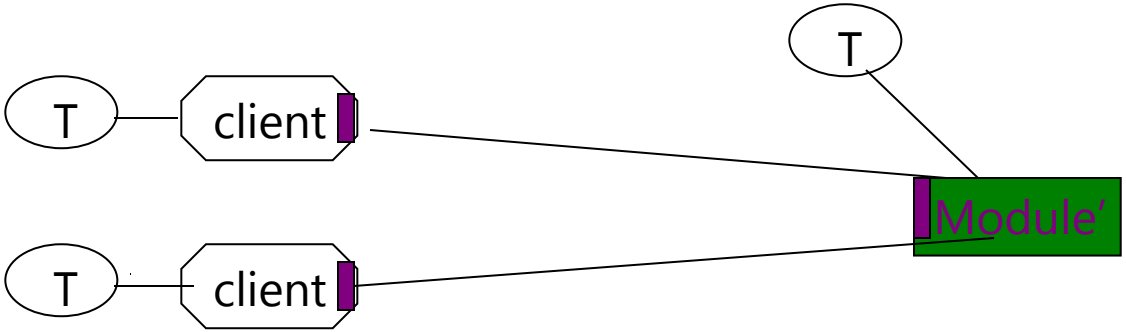

T
T
client
Module
T
client


T
T
client
T
client
Module
Module'


T
T
T
client
client
Module
Module'


T
T
T
client
Module
Module'
T
client


T
T
T
client
Module
Module'
T
client


T
client
T
client
T
Module'

How to make enhancement

- To enhance module M
  - Create M′
  - Move code from M to M′, making M call M′
  - Change interface and internal data structures of M′
    - Add tests for M′
    - Add variables to M to provide default values for parameters, change M to use new M′
  - Rewrite other modules to use M′ instead of M

Behavior preservation

- The reason we change program is usually to change behavior
- But we want to keep most of old behavior

- How can we change only the behavior we want?

Lessons from refactoring

- Focus on particular kind of change
  - Transformation becomes simpler – analysis becomes more shallow
- Keep programmer in control
- Support incrementalism

Challange

- “You can’t add in XX, you have to build it in from the beginning.”
- Show how to add on a particular software quality.
- Provide evidence for “It might be more expensive to do it later, but if it is worth it, you can always fix your software”.