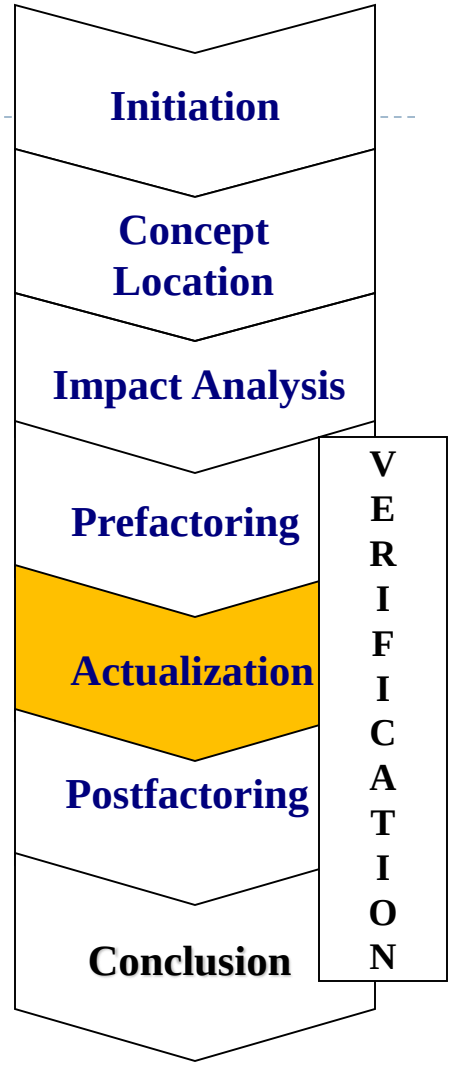
Actualization

Roberto A. Bittencourt
Based on Rajlich's slides

Actualization

- Programmers implement the new functionality
  - according to change request
- The process of actualization varies
  - depends on the size of the change



Small changes

- Done directly in old code

```
class Address
{
  public move();
  protected String name;
  protected String streetAddress;
  protected String city;
  protected char state[2], zip[5];
};
```

Small changes

- Done directly in old code

```
class Address
{
  public move();
  protected String name;
  protected String streetAddress;
  protected String city;
  protected char state[2], zip[9];
};
```

Small changes

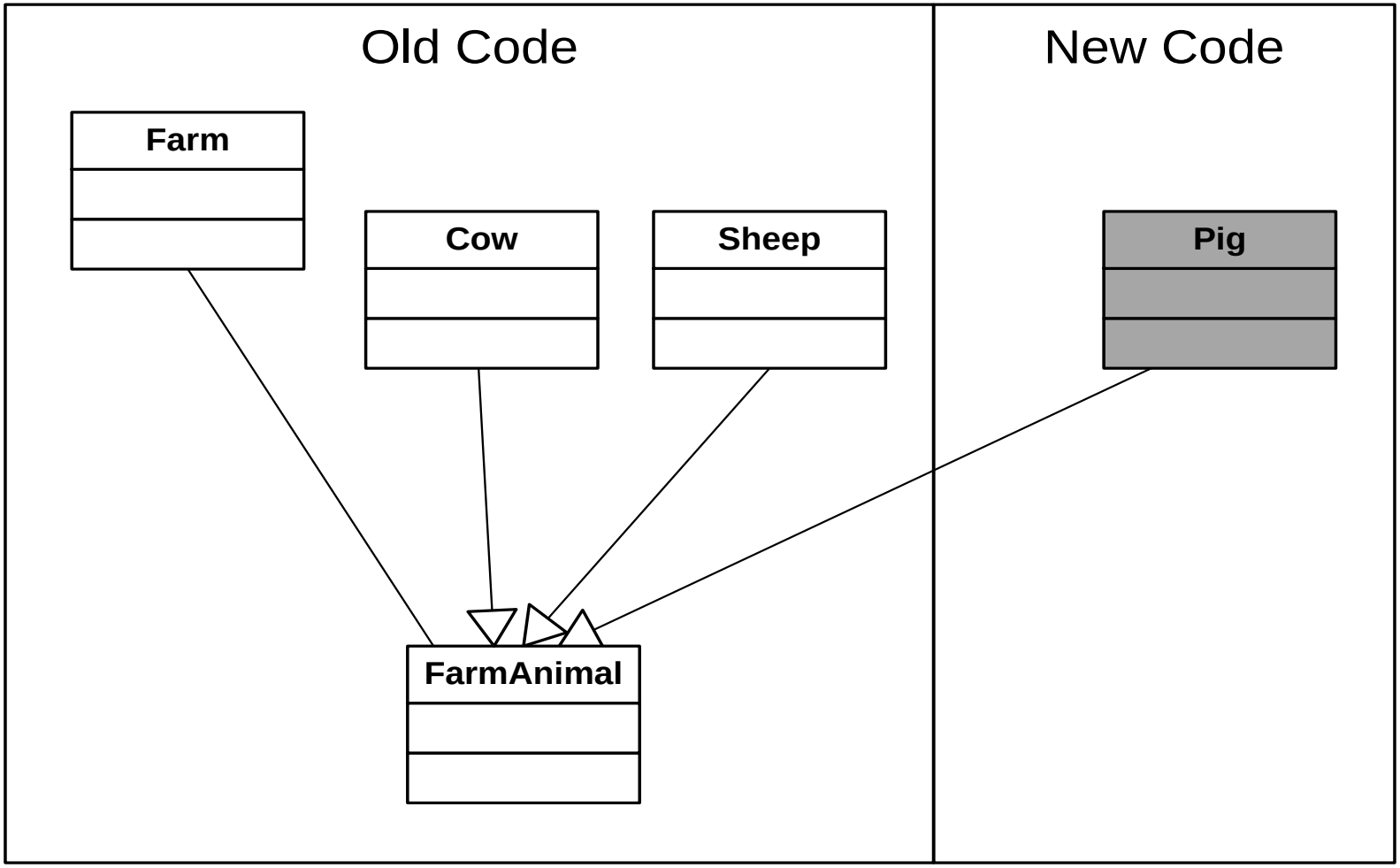
- Done directly in old code

```
class Address
{
  public move();
  protected String name;
  protected String streetAddress;
  protected String city;
  protected char state[2], zip[9];
};
```

Larger changes

- Programmers implement the new classes separately from the old code
- The new code is plugged into the the existing code
  - incorporation
- The change can propagate to other components of the system
  - ripple effect

Polymorphism



Polymorphic class

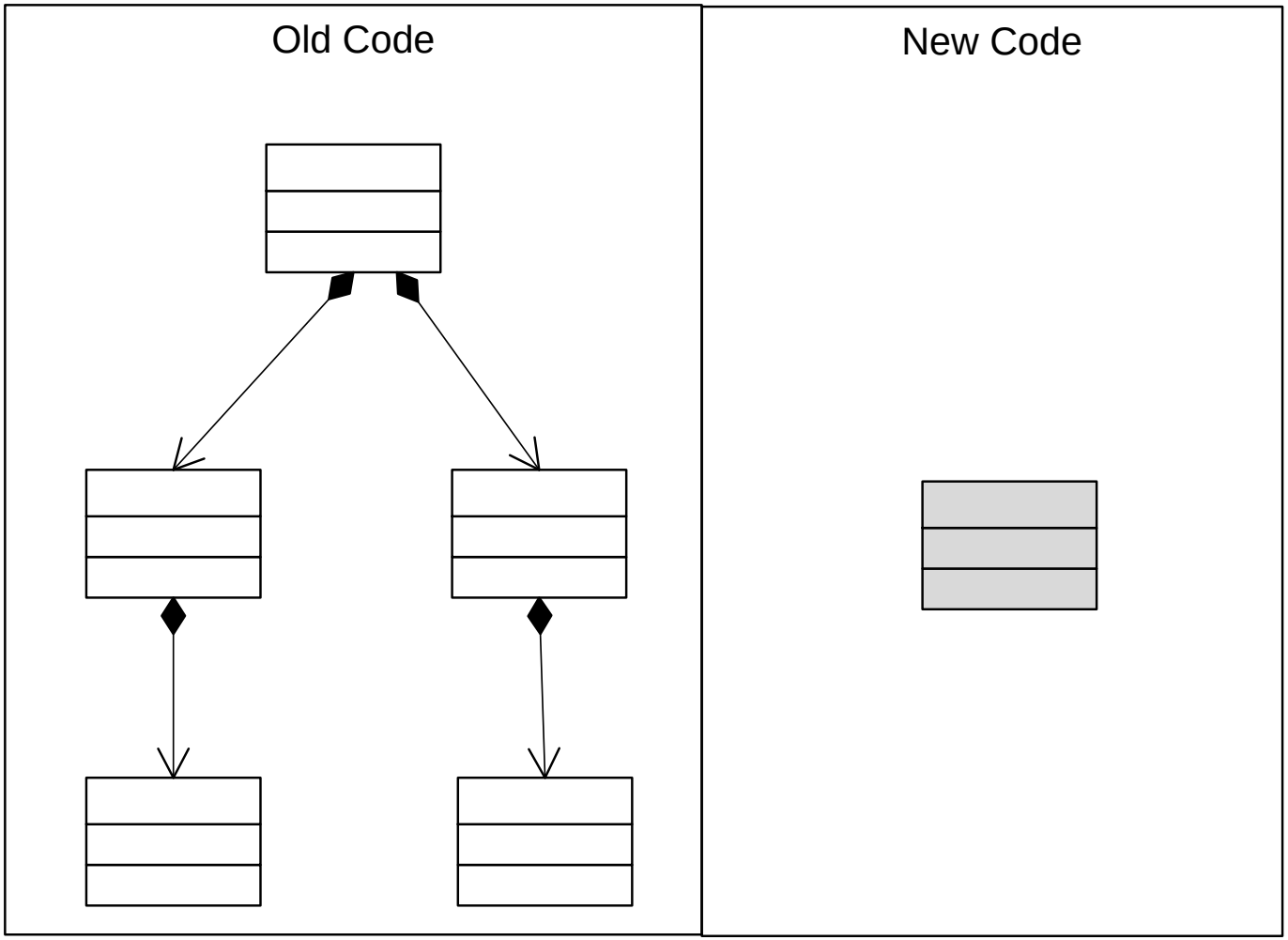
```
class Pig : public FarmAnimal
{
    public:
    void makeSound() {cout<<"Oink";}
};
```

- Farm now can declare objects of the type Cow, Sheep, or Pig
  - the composite responsibility of Farm was extended by the concept Pig.

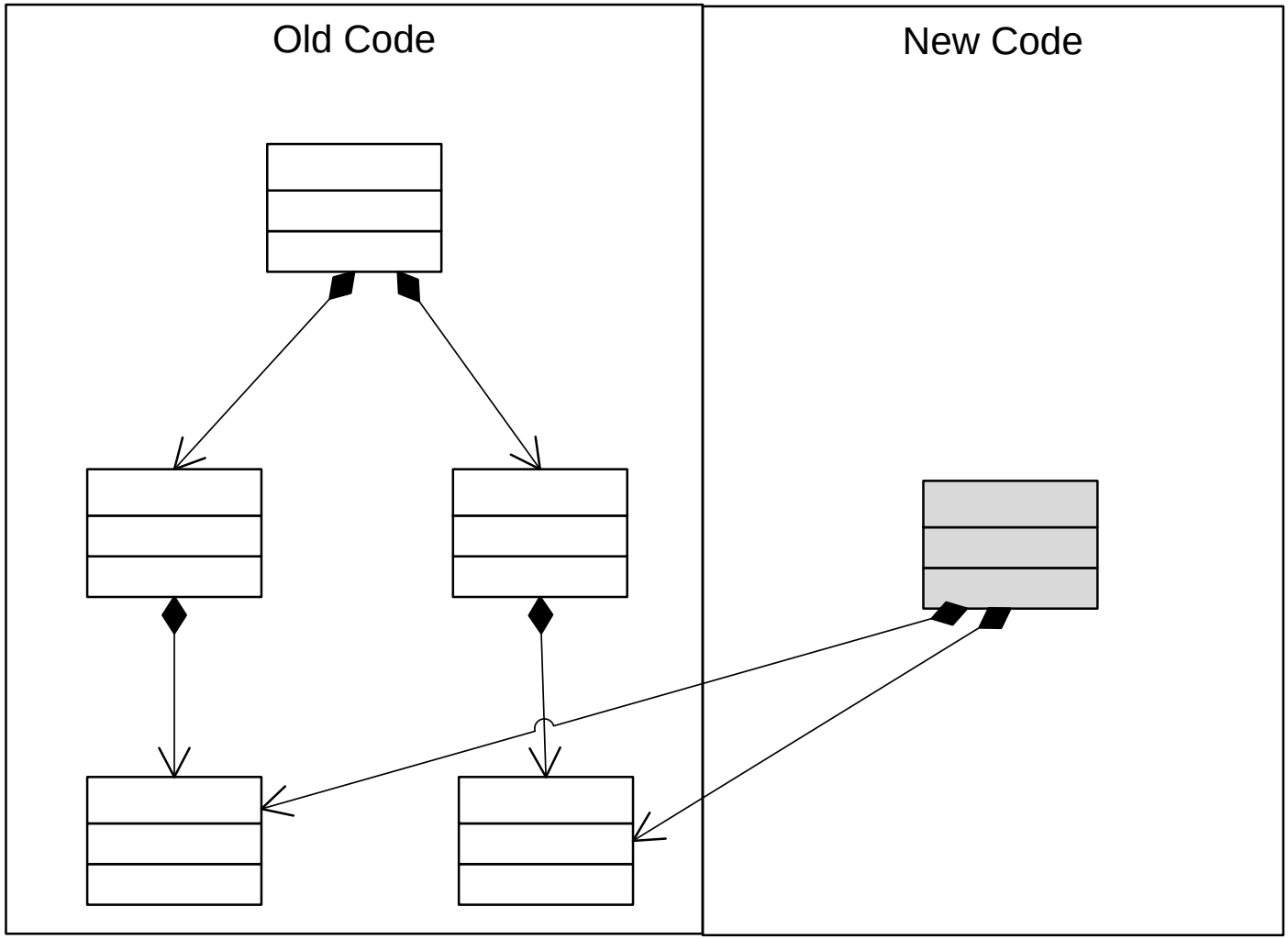
Adding New Component

- Implement the new classes separately from the clients in the old code
  - new classes assume the responsibilities demanded by the change request
- New classes are plugged as components into the appropriate place of the existing code
  - incorporation
- Change propagation

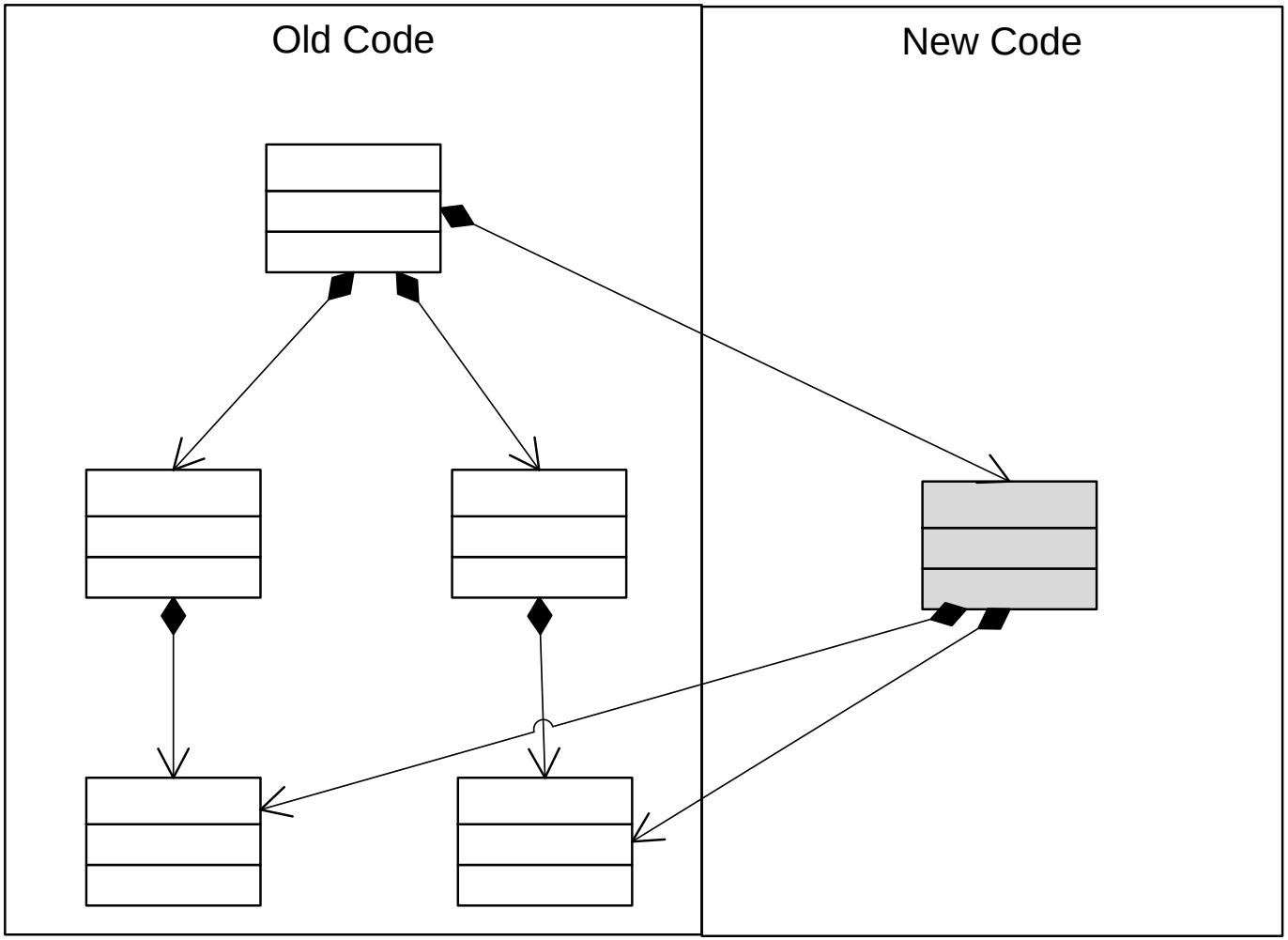
New responsibility is local



New responsibility is composite



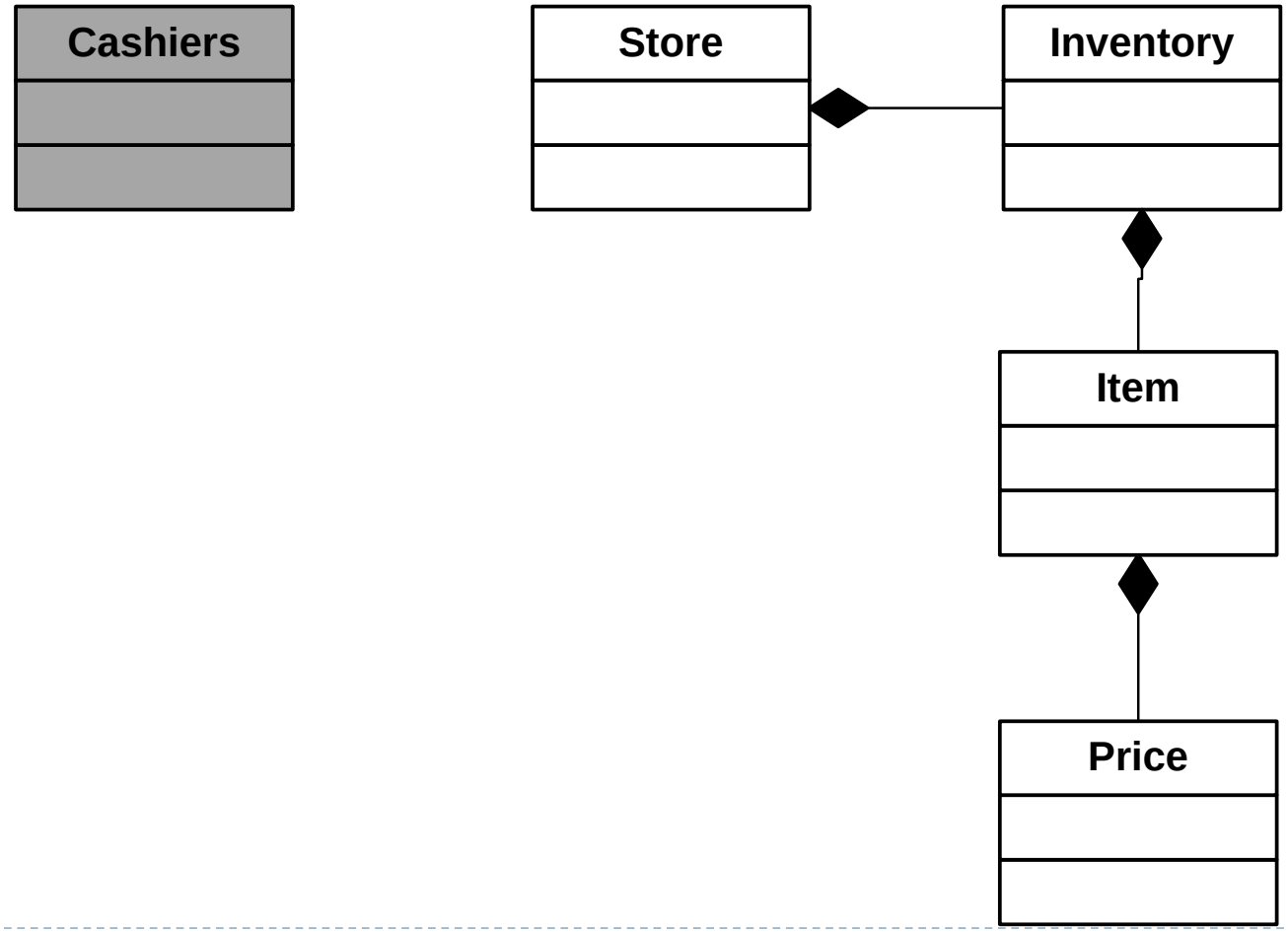
Incorporating new supplier



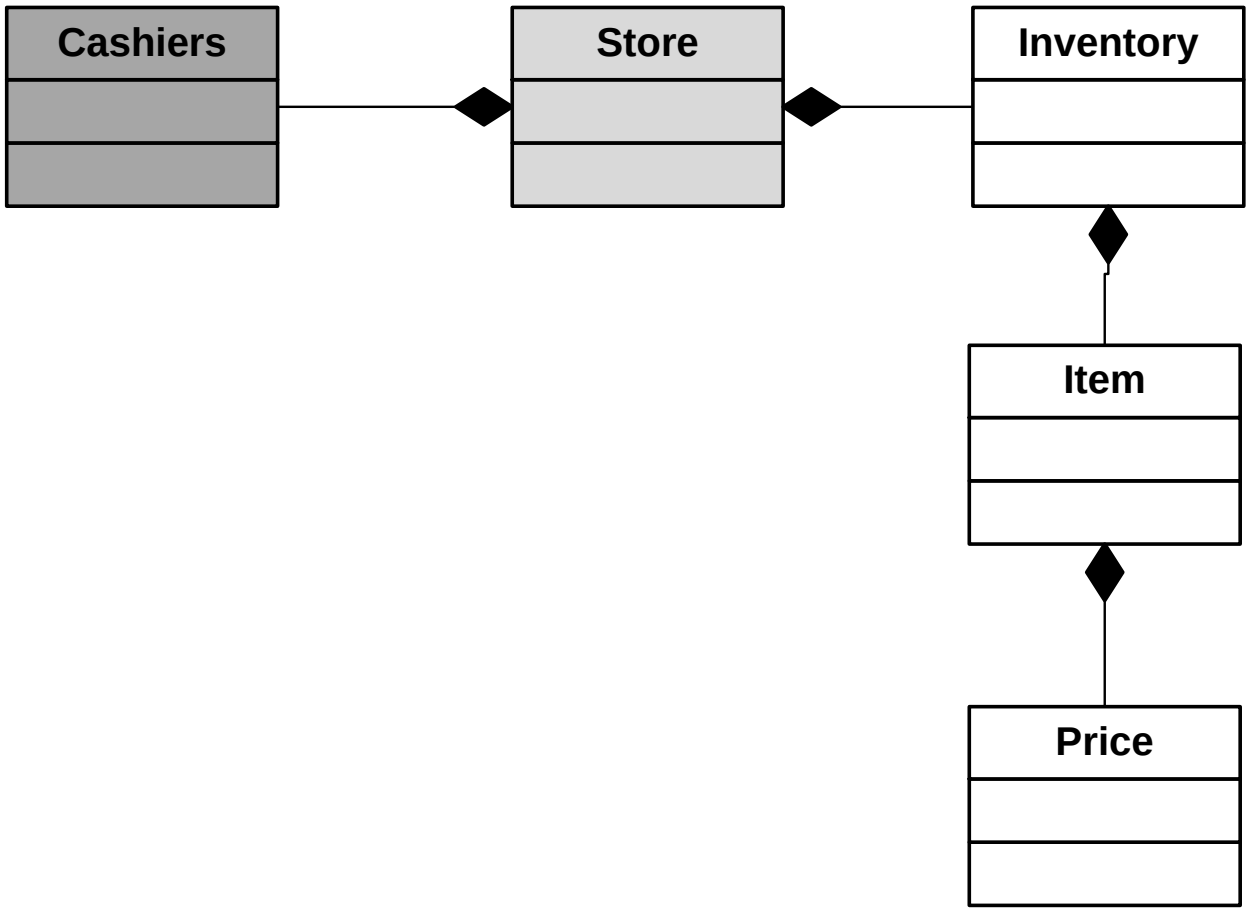
Point of Sale

- The old did not require authorization
  - anyone was able to launch the application
- The change request:
  - "*create a cashier login that will control the user log in with a username and password.*"

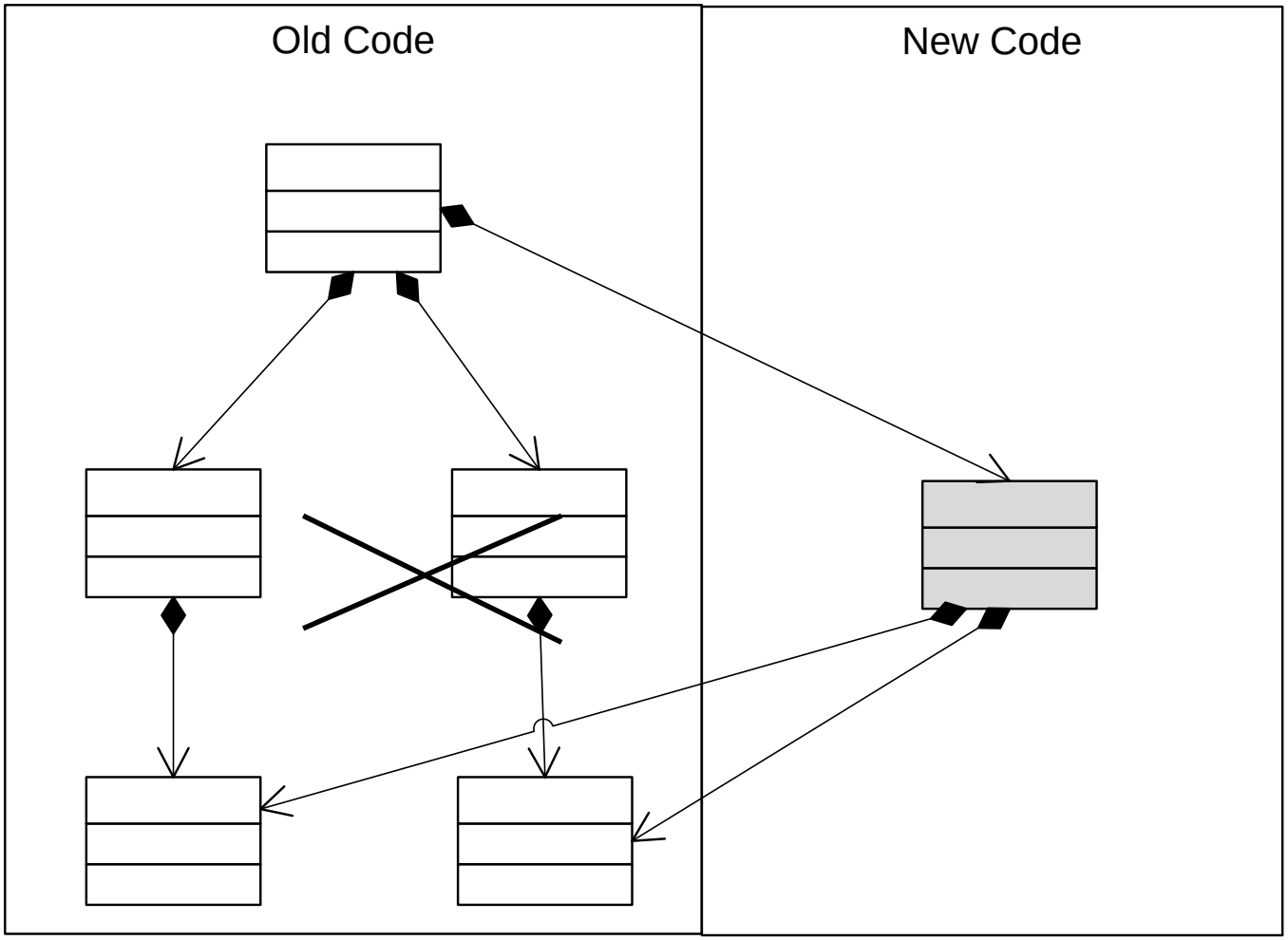
PoS + new class



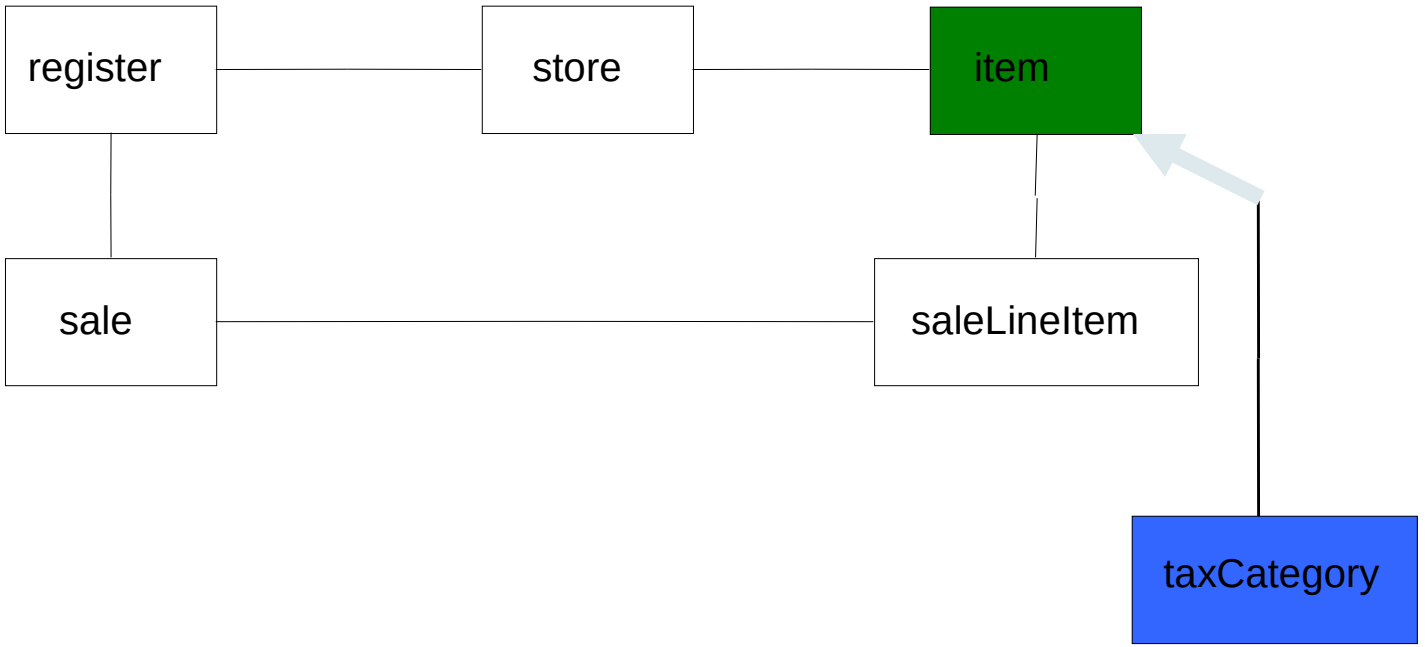
Incorporation of Cashier



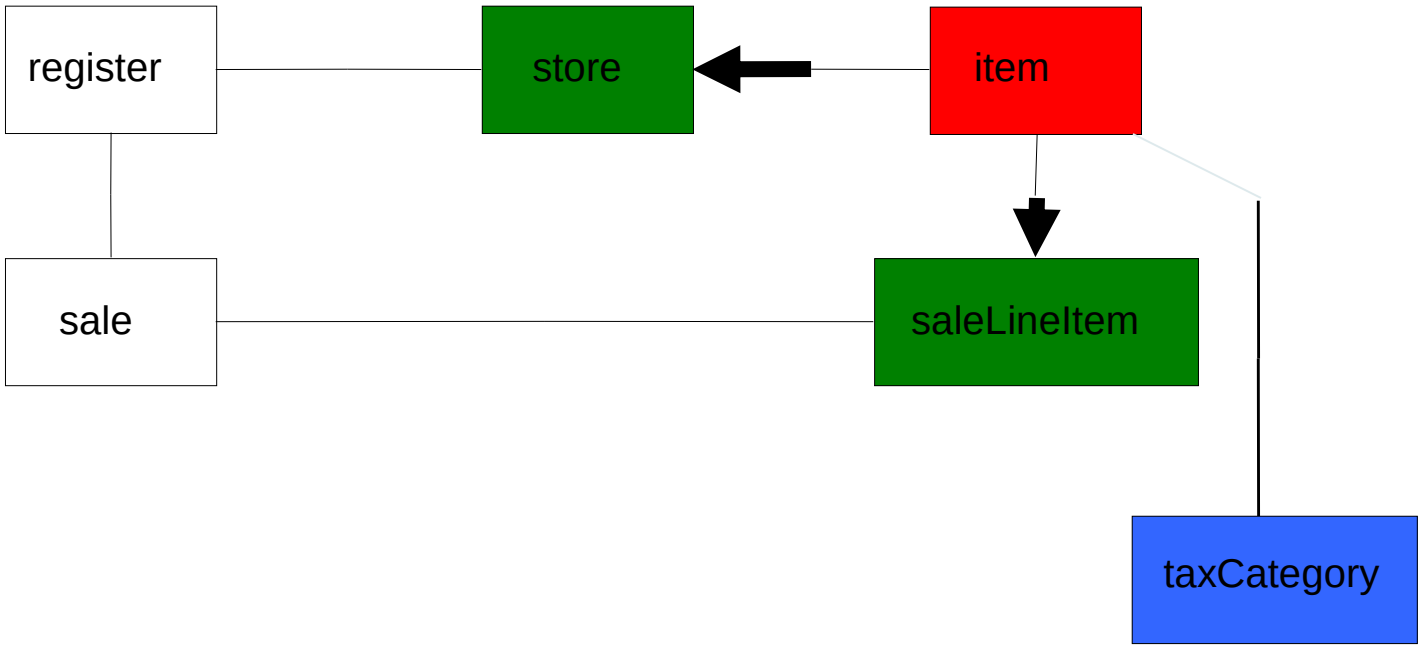
Replacement of a class



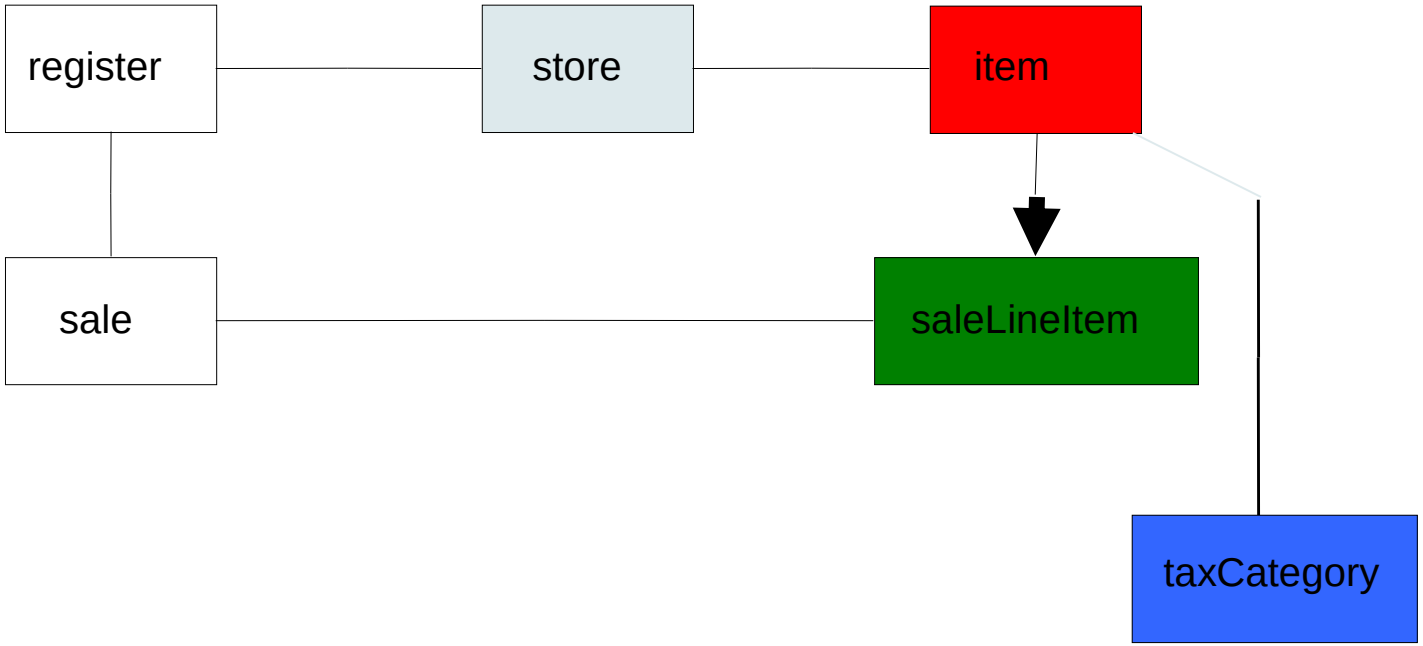
Example incorporation



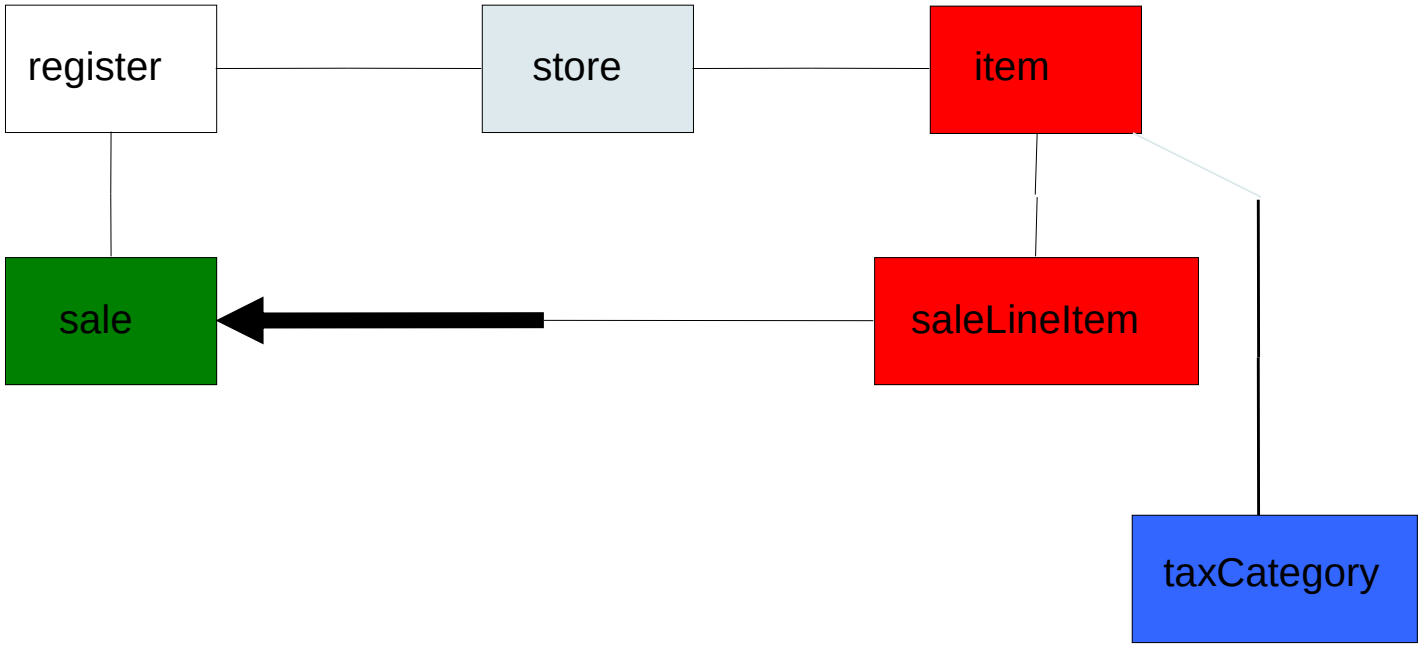
Change propagation



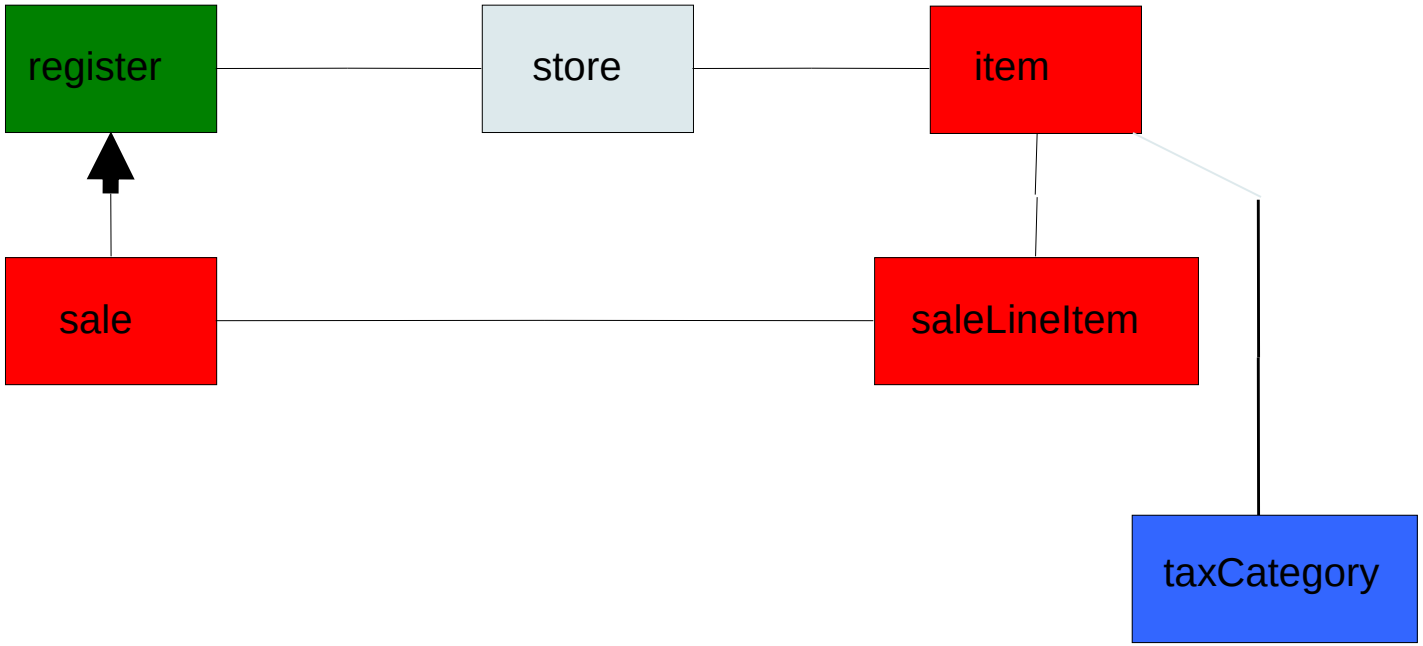
Change propagation



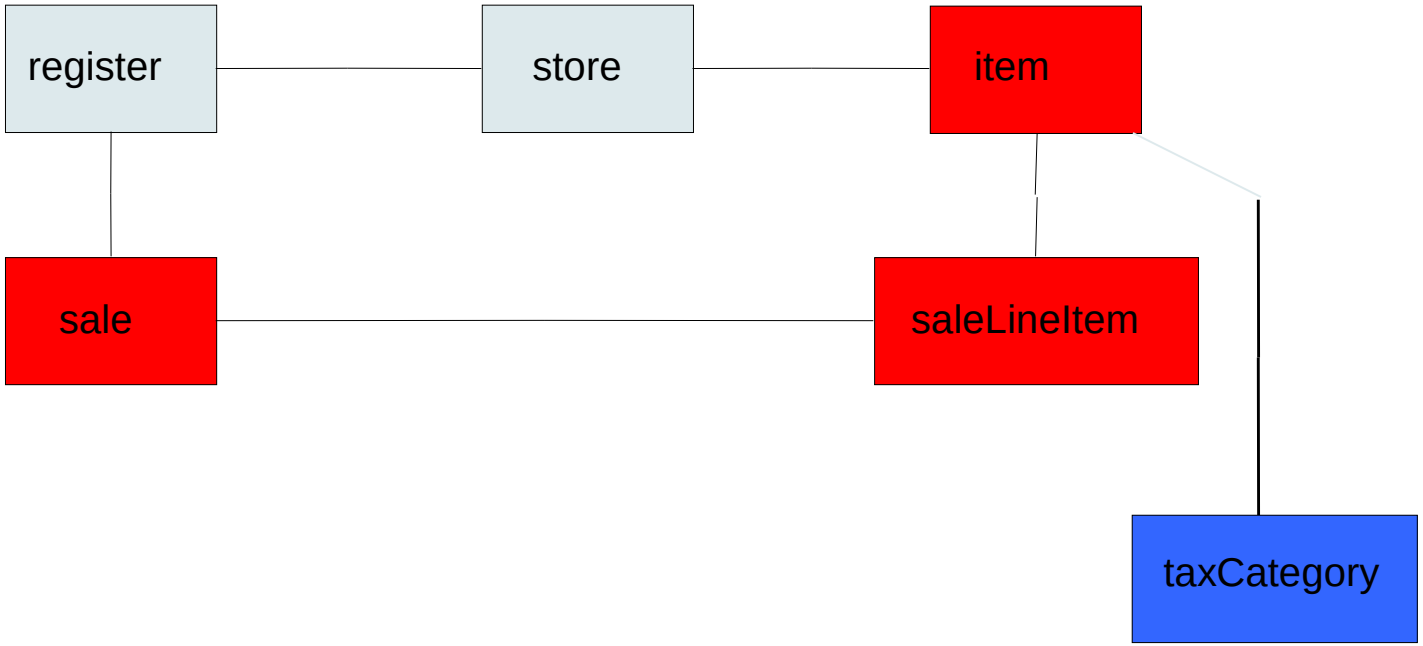
Change propagation



Change propagation



Change propagation ends



Deletion of obsolete functionality

- Also causes change propagation
- All references to the deleted functionality must be deleted
  - secondary changes propagate to other classes

Underestimated Impact Set

- Impact analysis estimates which classes are impacted
- Change propagation modifies the code of impacted classes
  - change propagation is the moment of truth
  - it confirms or refutes the predictions of impact analysis
  - accuracy of impact analysis predictions is important for software managers

Ericsson Radio Systems

		Predicted	
		Unchanged	Changed
Actual	Unchanged	42	0
	Changed	64	30

- total number of classes =

  42 + 0 + 64 + 30 = 136

Categories

- *true positives* = 30
- *false positives* = 0
- *true negatives* = 42
- *false negatives* = 64

Precision

- Used in the information retrieval
- Precision = (true positives)/(true positives + false positives)
- Ericson, precision = 30/(30 + 0) = 1 = 100%.

Recall

- Recall = (true positives)/(true positives + false negatives)
- Ericson recall = 30/(30 + 64) = 0.32 = 32%
- Programmers estimated that the changes will impact only about a third of all classes that actually changed
  - missed the other two thirds!

Underestimation

- Common in software engineering
  - consequence of invisibility
- Makes planning difficult
- Common in other field also