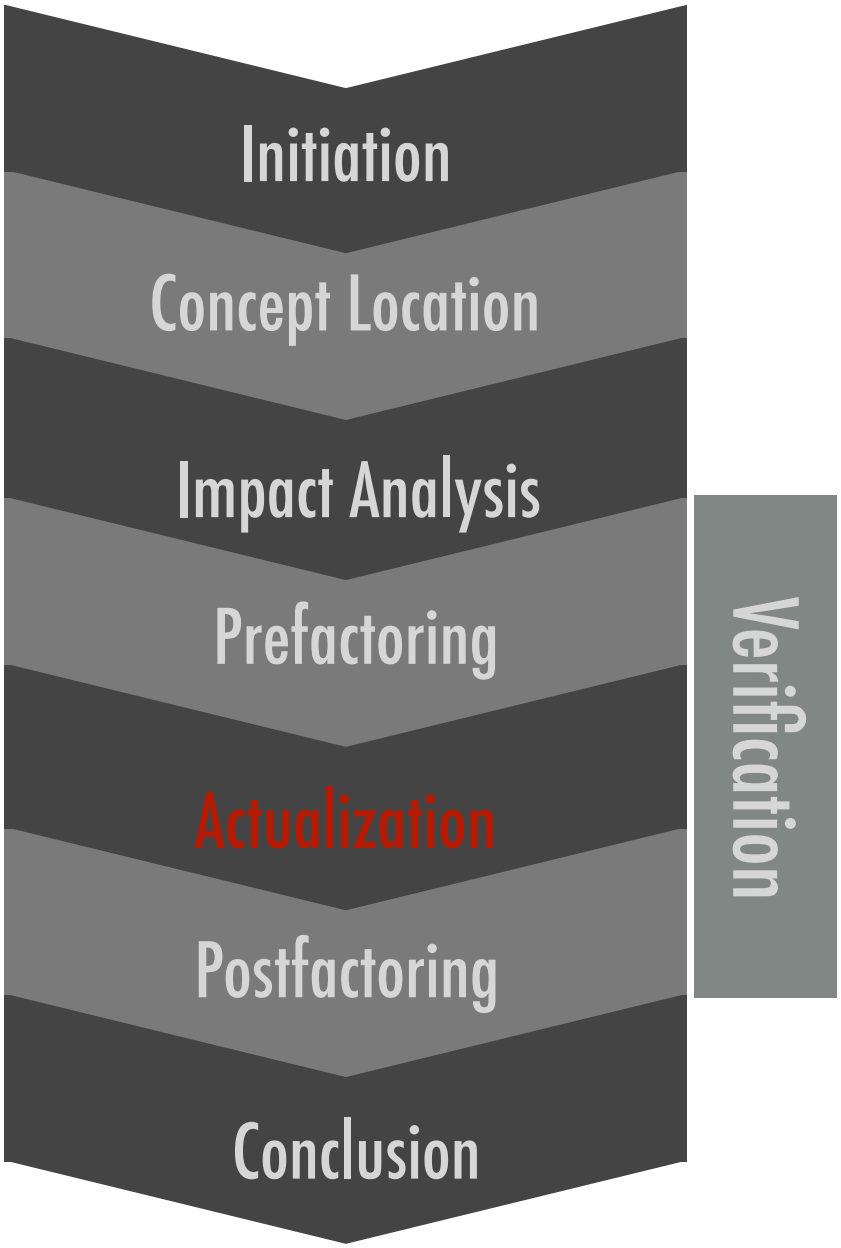
Actualization

Prof. Romain Robbes

Actualization adds new functionality



The programmer updates modules, adds new modules, and propagate changes to affected modules identified by IA.

Before changing the system, we should know how we will change it

We must select a change strategy.

Each change strategy has a different impact on the system.

Impact analysis depends on the selected strategy.

Outline

- Small changes vs large changes
- Change strategies
  - Through polymorphism
  - Adding new suppliers
  - Adding new clients
  - Replacement
- Change propagation
- Guidelines

Small changes

In some cases, it is enough to simply change a module

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

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

+ (eventual) propagation

This strategy is suitable for:

- small bug fixes
- localized increments of functionality

Still, different changes will have a different impact

Larger changes may require new modules

How do we integrate the new modules in the system?

Polymorphism

If the design anticipated the change, polymorphism may be an option

Polymorphism allows to add behavior without changing the interface

Change propagation can be minimal

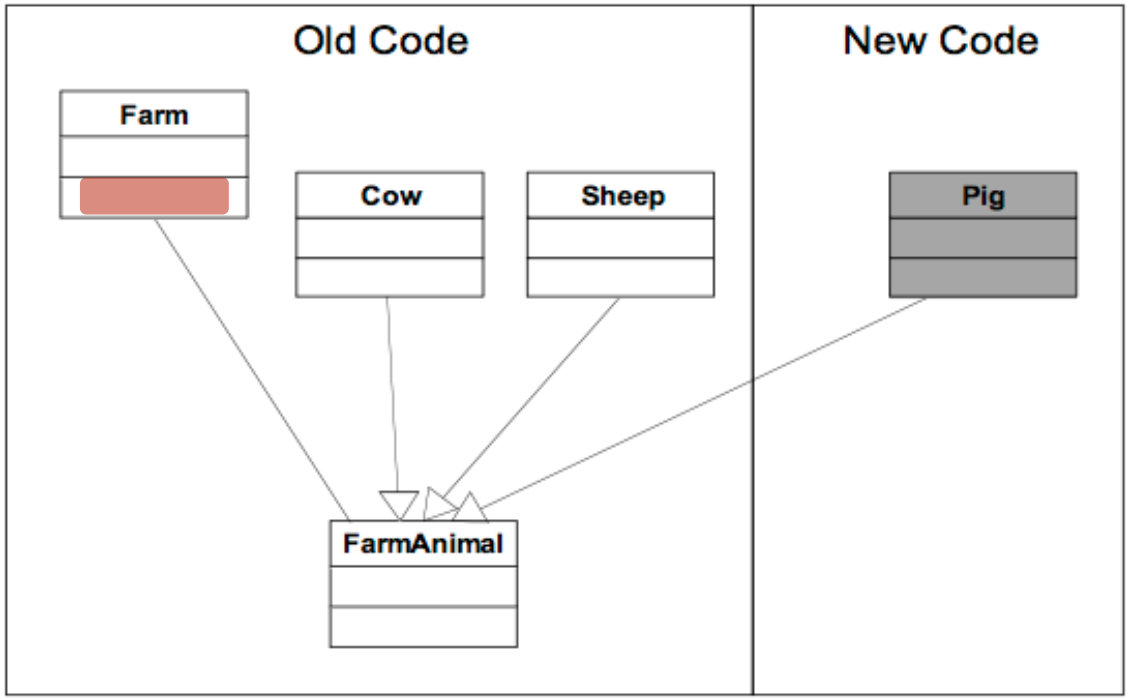
Example

```
class FarmAnimal {
   public abstract void makeSound();
};

class Cow extends FarmAnimal {
   public void makeSound(){
    System.out.println(”Moo-oo-oo”);
   }
}
class Sheep extends FarmAnimal {
   public void makeSound(){
    System.out.println(”Be-e-e”);
   }
}


class Pig extends FarmAnimal {
    public void makeSound(){
      System.out.println(”Oink”);
    }
}
```

The new code is easy to integrate



Of course, we still need to create new instances of class Pig ...
so usually additional changes are needed.

Anticipation of change is important

The design should have anticipated:

- that the family of concepts is likely to be extended
- how it is likely to be extended
  (so that the interface does not change)

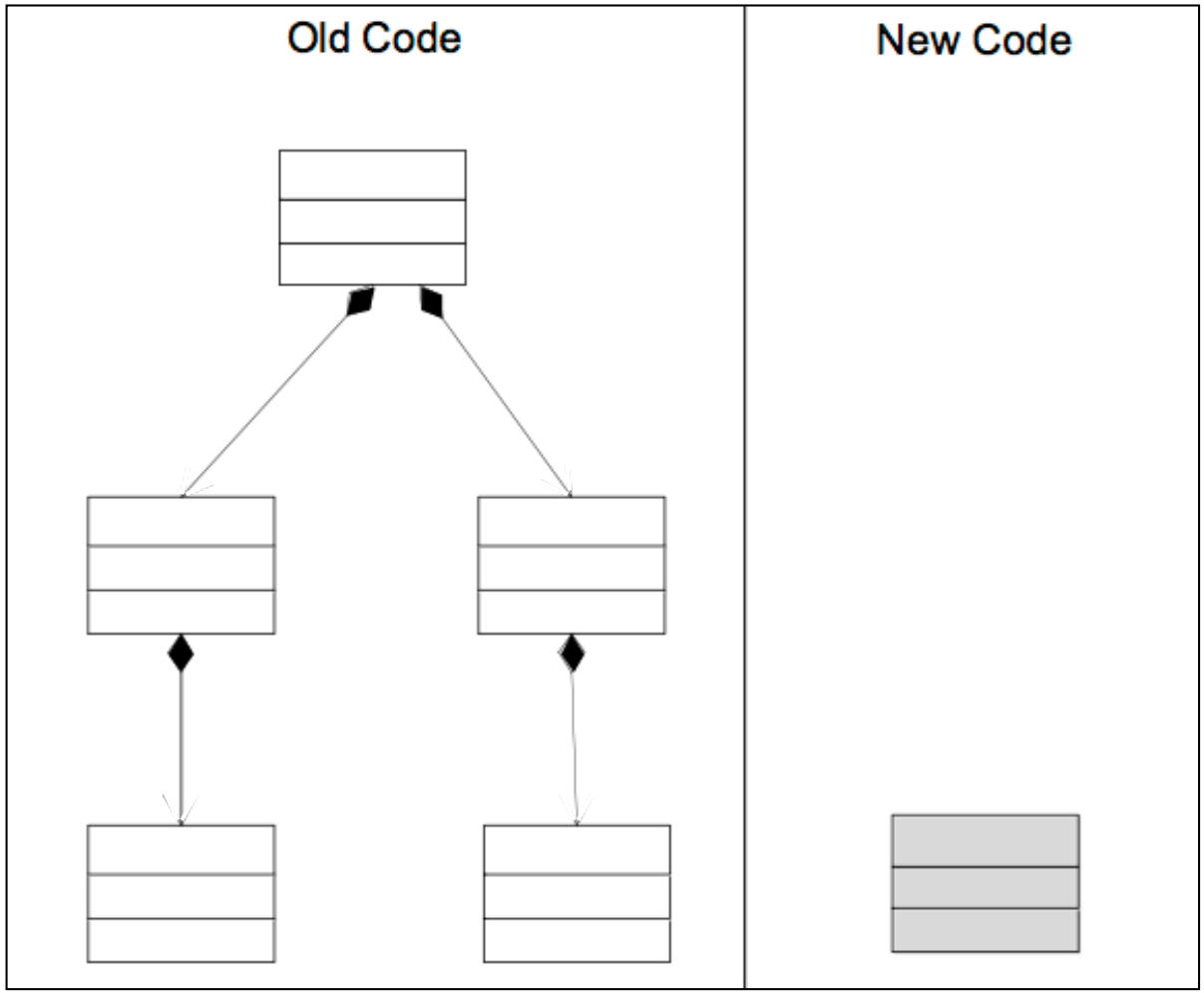
Adding new suppliers

Polymorphism is not always available–or appropriate.

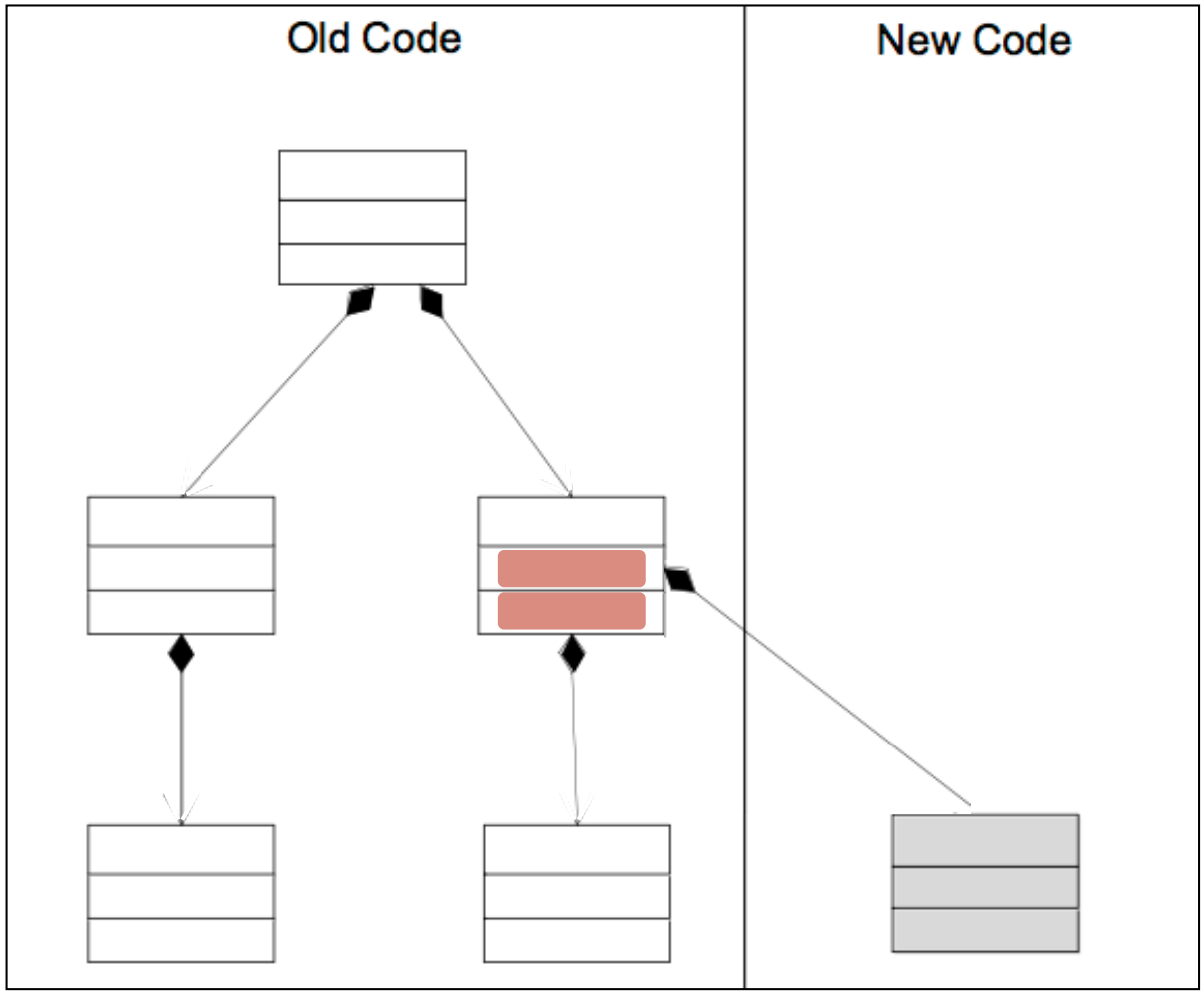
Genuine new functionality can not be added by refining older concepts.

New, standalone modules must be created and incorporated.

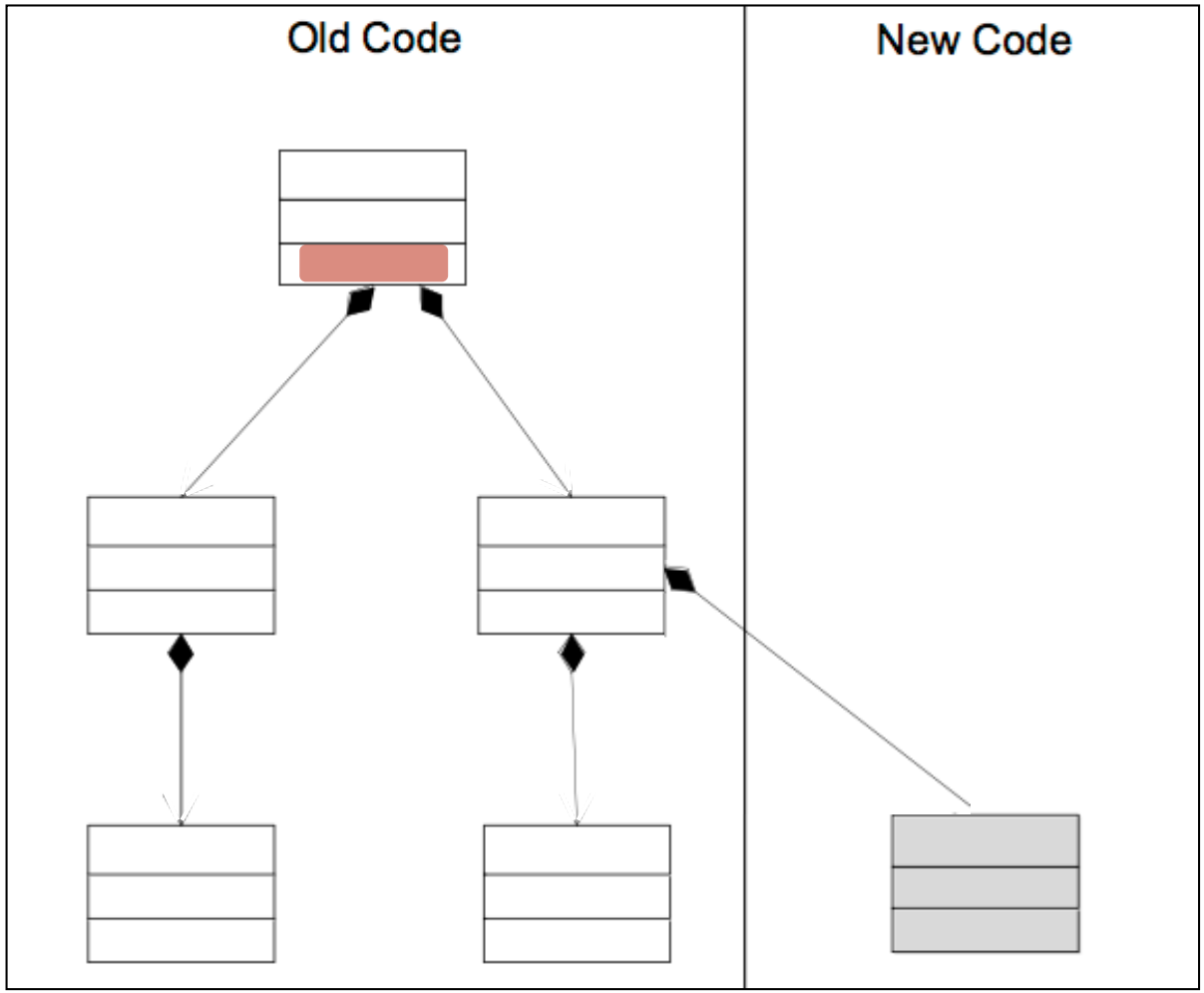
The new functionality is developed through a standalone module (or more)



The new functionality is integrated in the location identified by concept location

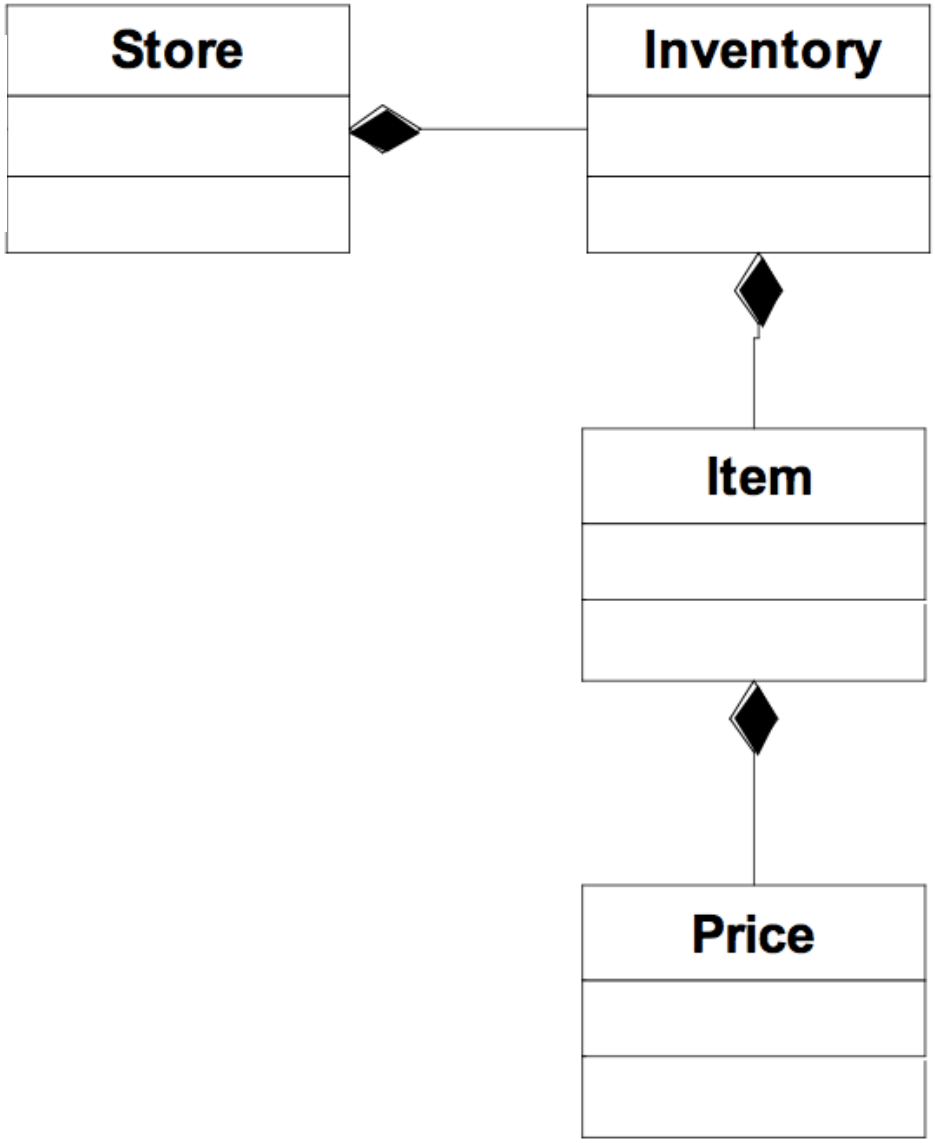


Then the changes are propagated

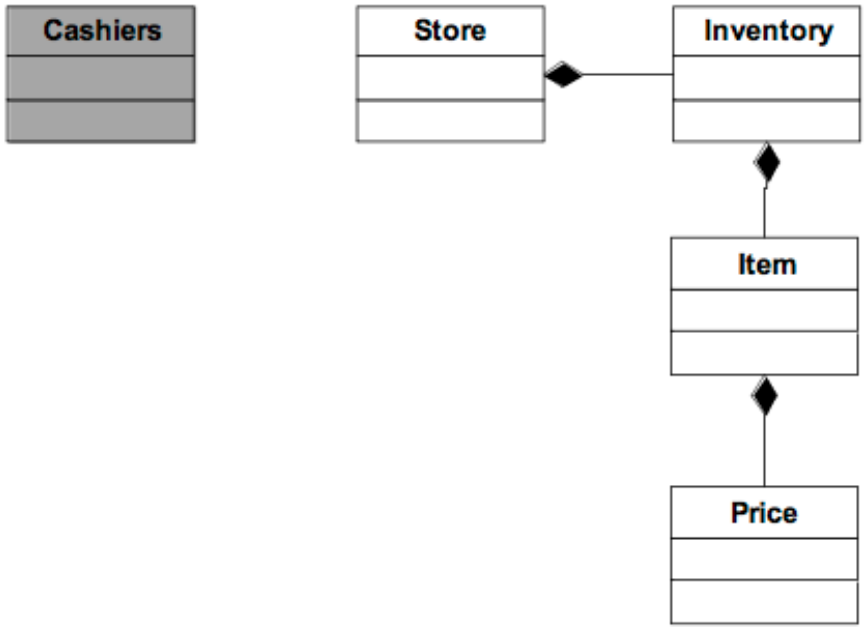


Example: change request for a point-of-sale system

"Create a cashier login that will control the user log in with a username and password."

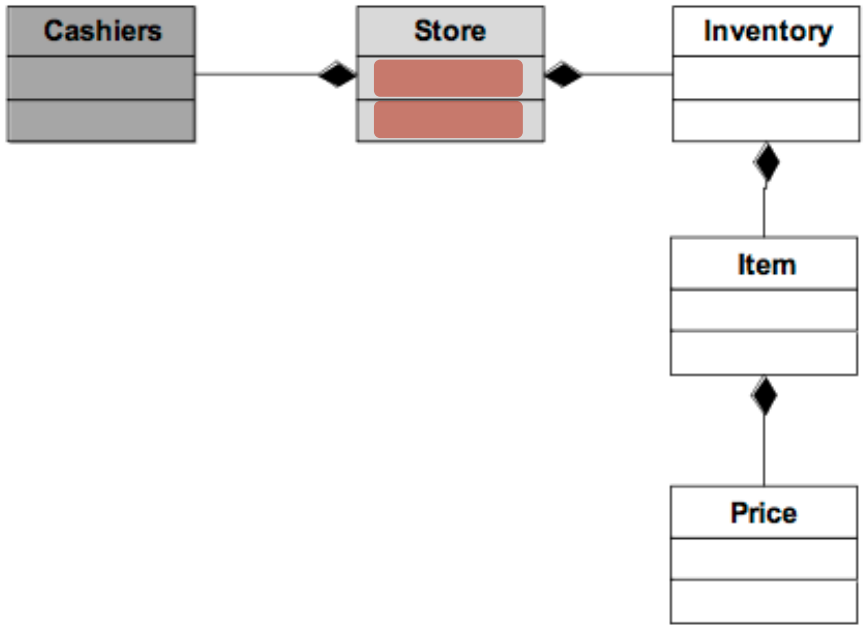


Implementing the standalone module



create new class `Cashier`, with fields `cashierID` and `password`; with methods `login()` and `logout()`

The new module is integrated in Store



add an instance variable to `Store`, and propagate changes (e.g. call `login()` when the application starts, `logout()` when it exits)

Incorporating new suppliers is more complex than using polymorphism

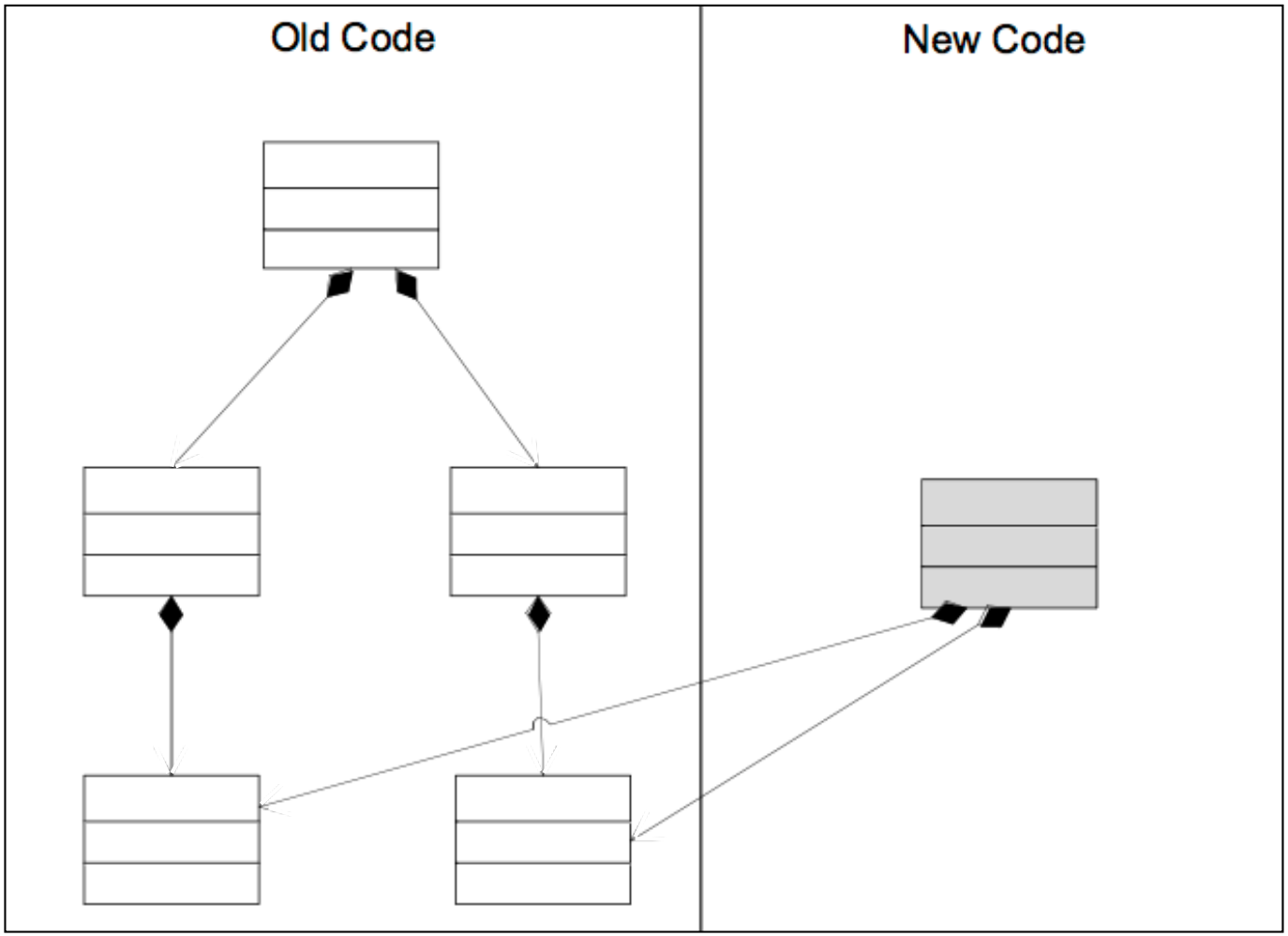
The change propagation may be larger, since the new functionality is not as easy to integrate in the system.

Adding new clients

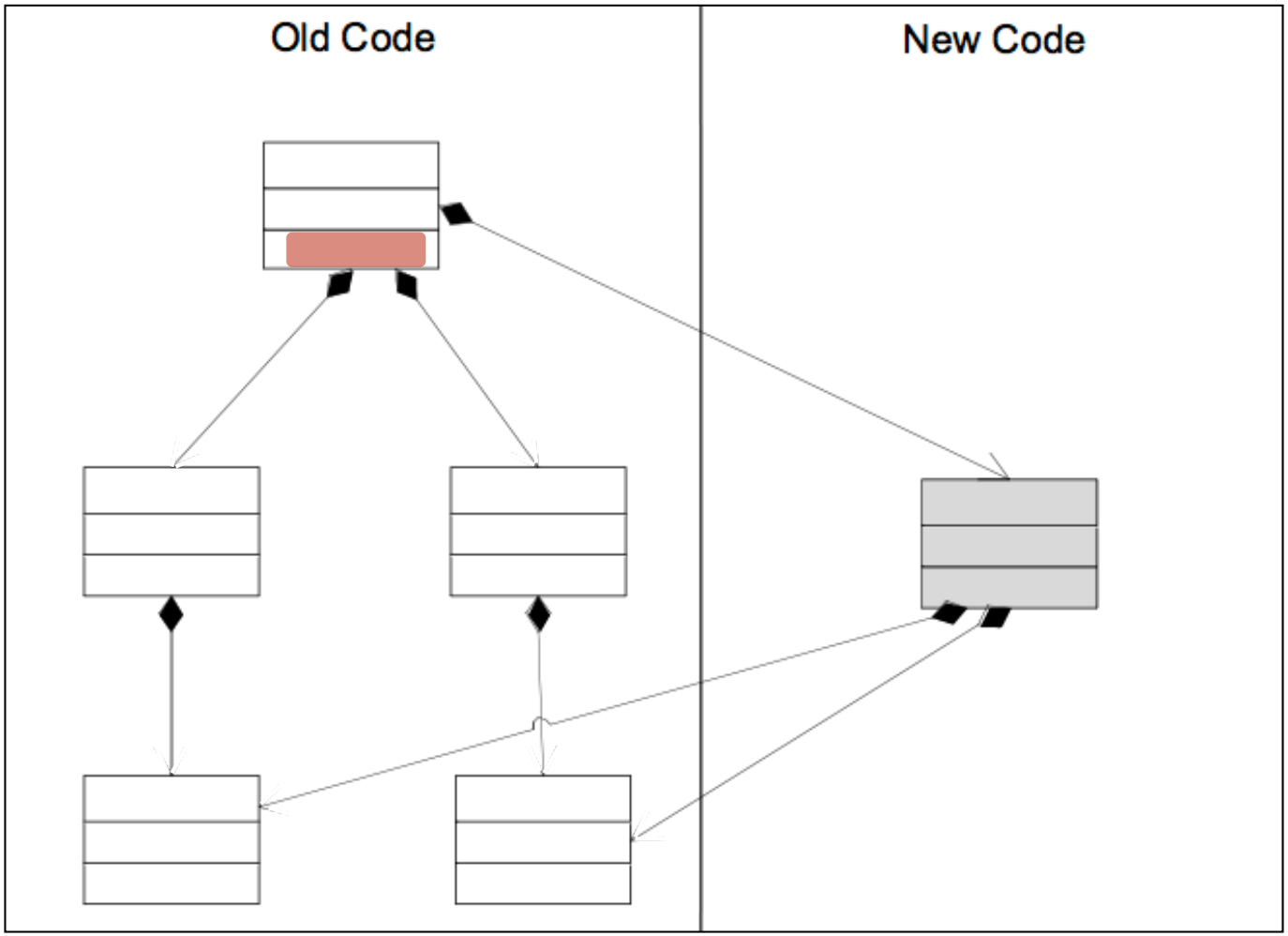
Sometimes the new functionality is a composition of existing functionality

We just need to “glue the pieces together”

First the new client, reusing old functionality, is developed



Then it is integrated in the application, using the location identified by CL

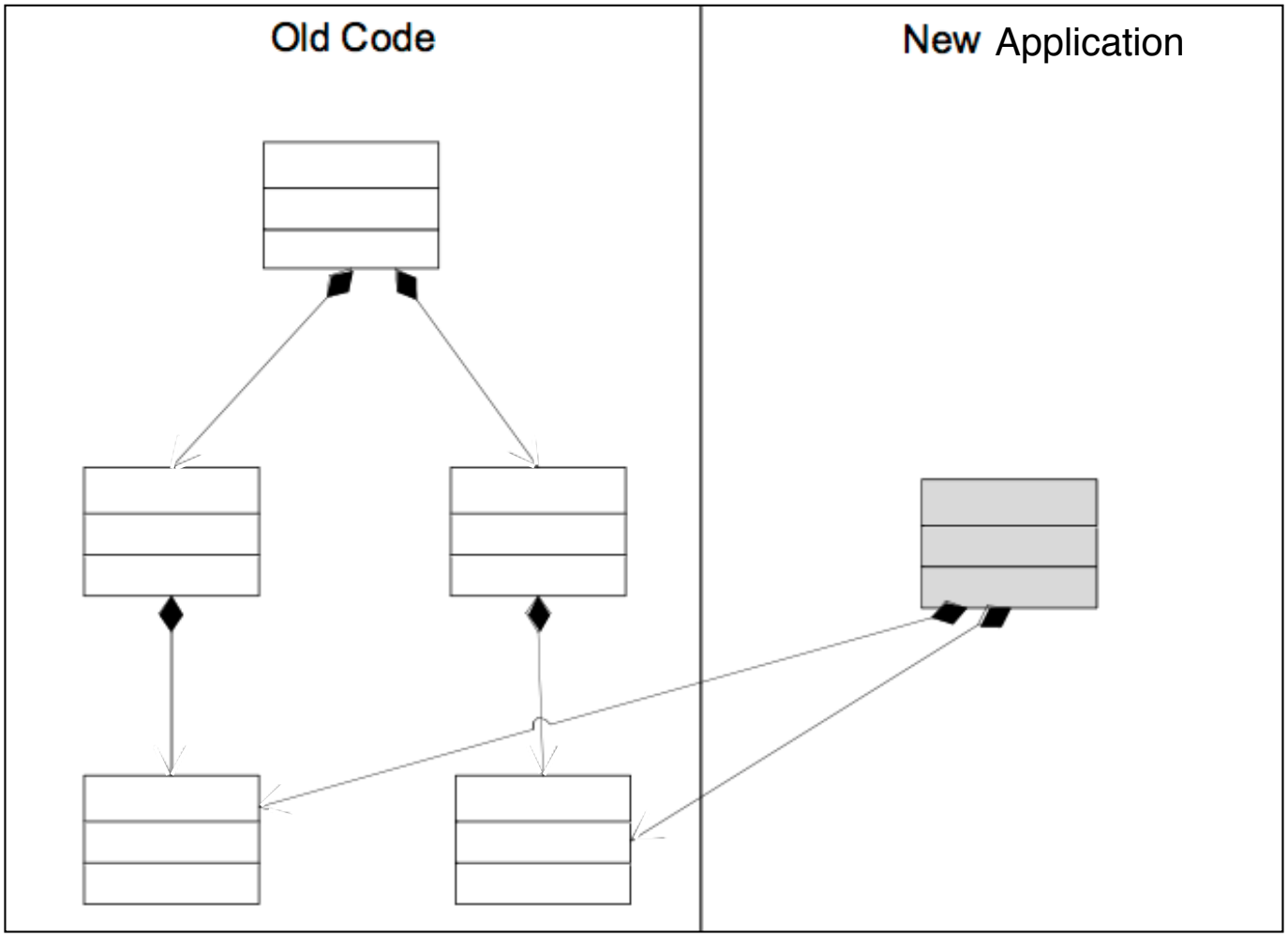


Example: adding reports to the store application

- Add an InventoryReport class, a new client of Inventory

- InventoryReport is also a new supplier of Store, the top module of the application

Sometimes, the new client is a top-level module, making a new application



e.g., an employee management application independent from the Store

As in the case of adding suppliers, change propagation may be large

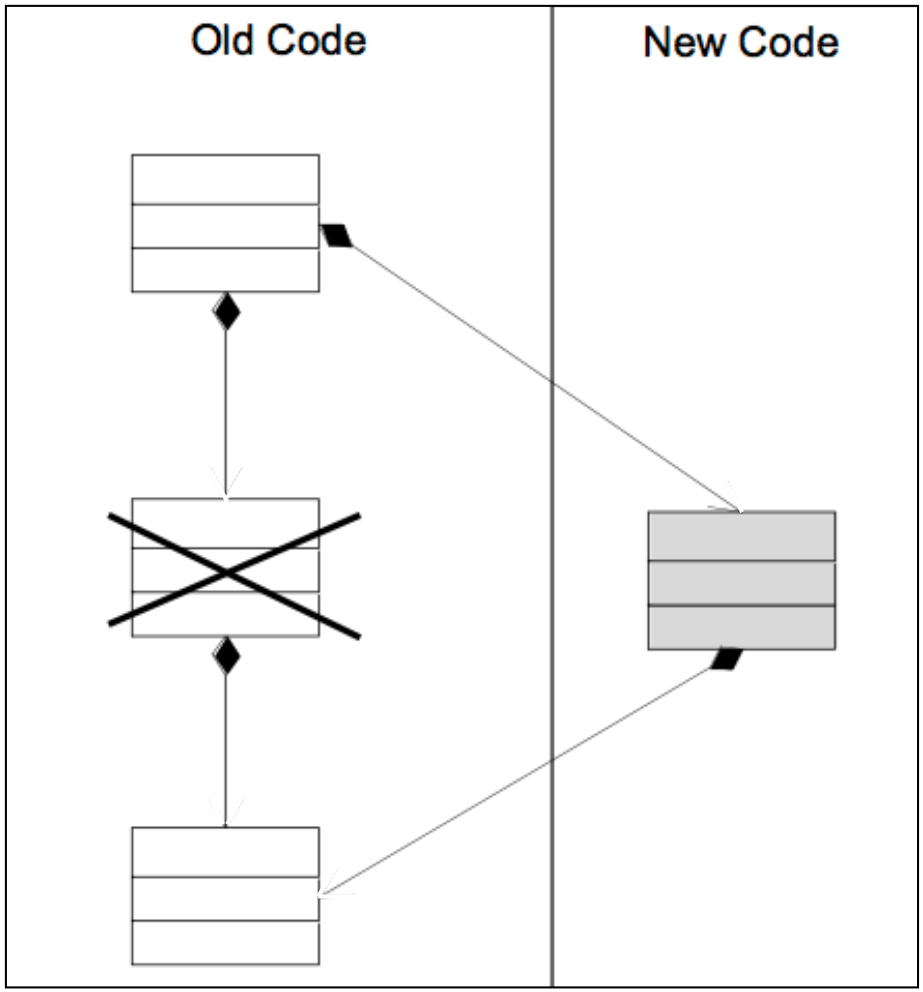
At least, we reused code: the implementation of the new functionality is less than the other case.

Module replacement

Obsolete functionality needs to be replaced

If the existing functionality is no longer useful after the change, we can replace it with the updated version.

References to the old module are replaced with references to the new one



Example: integrating promotions in the point-of-sale application

Change request: "support price fluctuations; the users of the program should be able to set prices of items in advance and be able to adjust the item prices on selected dates. For example, there can be sales periods where on certain dates the price is lower, while after these dates it returns to the previous level."

Example: integrating promotions in the point-of-sale application

Parametrizing prices with dates is a complex change. Class `Price` is a very simple data class (e.g. `getPrice()`, `setPrice(...)`); we need to totally replace it!

Change propagation

Once the change is performed, we need to propagate it

Changes in the interface break the interactions of modules using it.

The changes must be processed one by one.

The process is similar to IA, but actual changes are performed.

JRipples provides support for change propagation too.

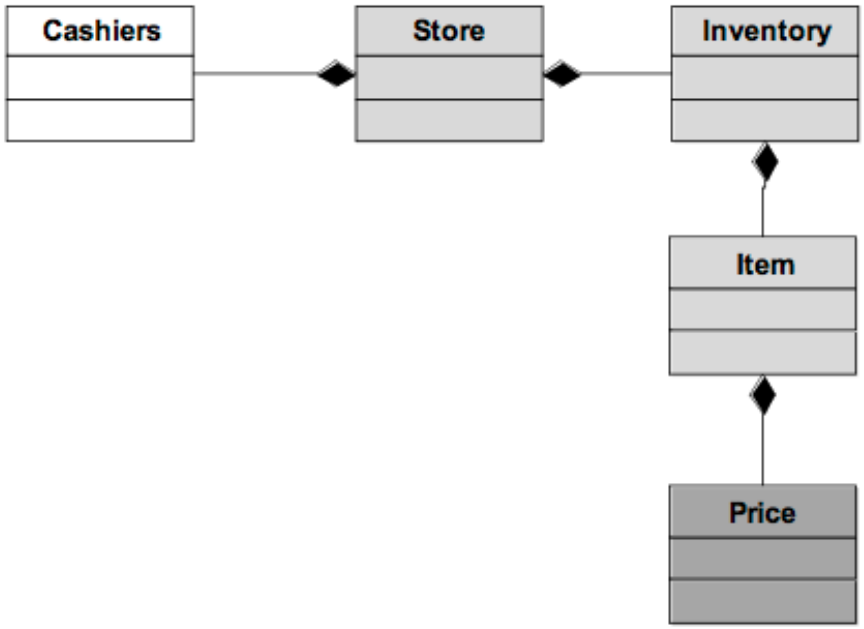
Each modification done during change propagation is usually small

It can be done in place (see: small changes).

Usages of obsolete functionality need to be removed as well

or converted to the new functionality.

Changing the interface of class Price (adding a date) propagates in the system



Possible reason: Item is not the right place to get a value for the date, neither is inventory.

The changed modules constitute the changed set; compare to the impact set

Programmers tend to underestimate the changed set.
This is a consequence of the invisibility of software.

Example at Sony-Ericsson:

		Estimated	
		Unmodified	Modified
Actual	Unmodified	42	0
	Modified	64	30

False Positives

True Negatives

False Negatives

True Positives

Precision = TP / (TP + FP) = 100%

Recall = TP / (TP + FN) = 32%

If the old and the new functionality can coexist, an alternative is deprecation

Constructor Summary

`Date()` Allocates a `Date` object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.
`Date(int year, int month, int date)` **Deprecated.** *As of JDK version 1.1, replaced by* `Calendar.set(year + 1900, month, date)` *or* `GregorianCalendar(year + 1900, month, date)`.
`Date(int year, int month, int date, int hrs, int min)` **Deprecated.** *As of JDK version 1.1, replaced by* `Calendar.set(year + 1900, month, date, hrs, min)` *or* `GregorianCalendar(year + 1900, month, date, hrs, min)`.
`Date(int year, int month, int date, int hrs, int min, int sec)` **Deprecated.** *As of JDK version 1.1, replaced by* `Calendar.set(year + 1900, month, date, hrs, min, sec)` *or* `GregorianCalendar(year + 1900, month, date, hrs, min, sec)`.
`Date(long date)` Allocates a `Date` object and initializes it to represent the specified number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT.
`Date(String s)` **Deprecated.** *As of JDK version 1.1, replaced by* `DateFormat.parse(String s)`.

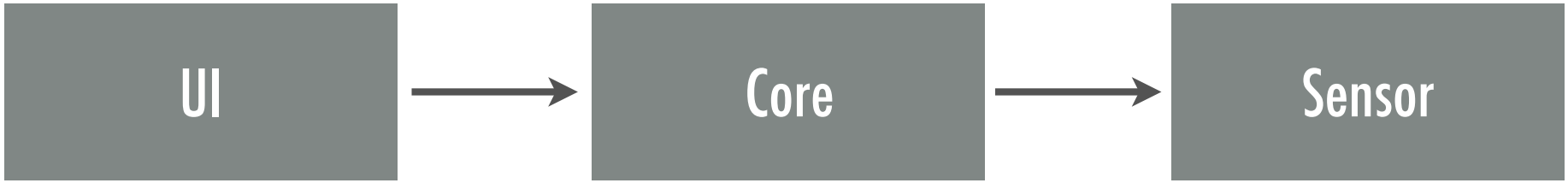
Uses of the old functionality are not replaced, but a deprecation message is shown. This allows clients to migrate gradually to the new interface. This is used a lot in APIs, where the stability of the clients is important.

Guidelines

Where to implement the change?

Each solution will have different characteristics

Example: display temperature in ~~Fahrenheit~~ Celsius



1. Change in the core: add conversion function
2. Change in the UI: do the conversion before display

2. is easier, but makes future maintenance harder.
This is decided in the IA phase.

Which change strategy to adopt?

Strategy	Extant of propagation	Characteristics
Small changes	small to large	localized change
Polymorphism	small	anticipated by design
Adding suppliers	medium to large	brand new functionality
Adding clients	medium to large	reuse and compose old functionality
Replacing modules	medium to large	old functionality is obsolete
Deprecation	none	old and new functionality can coexist

Of course, anticipating changes is the best strategy

In all cases, small (cohesive, low-coupled) modules and classes with well-defined interfaces are helpful:

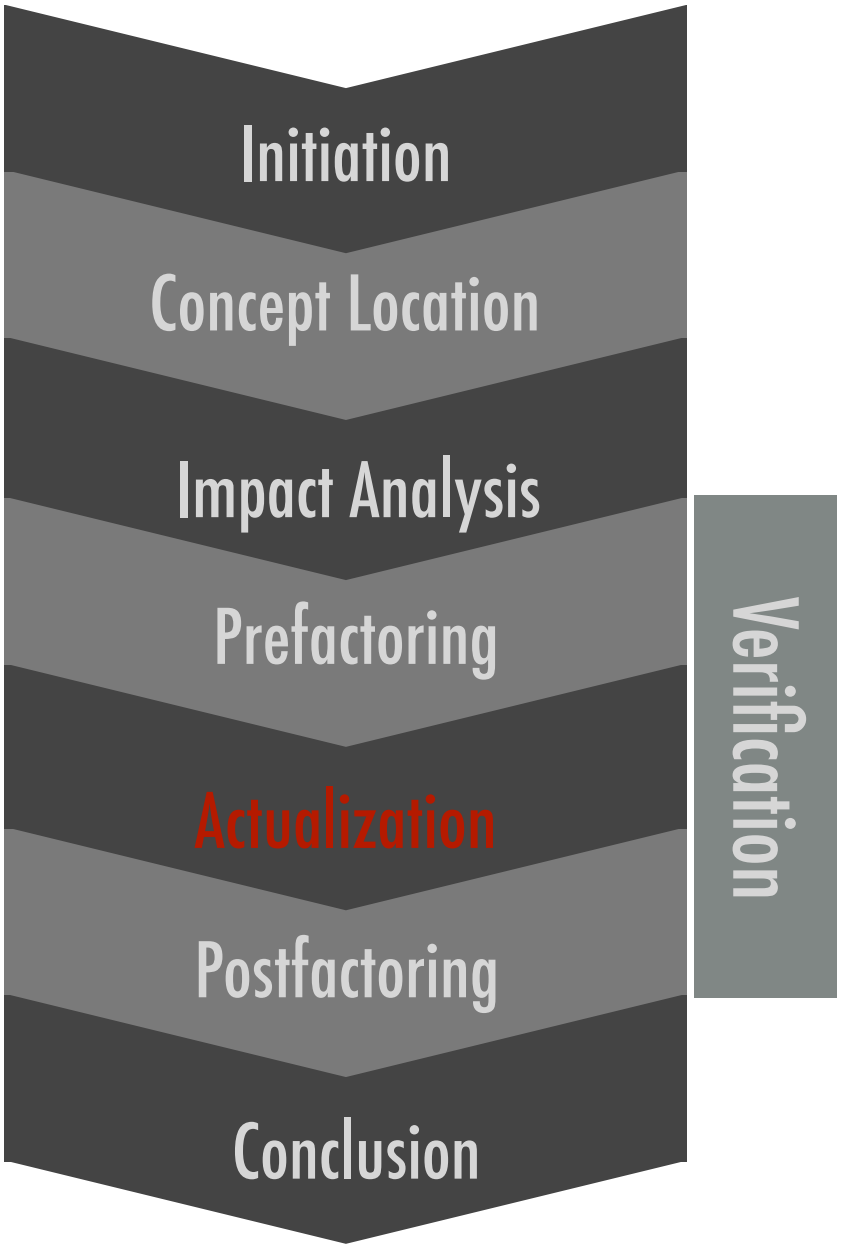
- easier to extend (including polymorphism)
- easier to replace
- easier to compose

Some design patterns help anticipating change

e.g. Strategy, Visitor, Decorator, Observer ...

Conclusions

Actualization adds new functionality



The programmer updates modules, adds new modules, and propagate changes to affected modules identified by IA.

The change strategy is chosen during IA, and implemented here

Strategy	Extant of propagation	Characteristics
Small changes	small to large	localized change
Polymorphism	small	anticipated by design
Adding suppliers	medium to large	brand new functionality
Adding clients	medium to large	reuse and compose old functionality
Replacing modules	medium to large	old functionality is obsolete
Deprecation	none	old and new functionality can coexist