

*Un-anticipated  
customization  
of software  
products*


*Sebastien Vaucouleur  
vaucouleur@itu.dk  
IT University of Copenhagen*



# Context and contributors

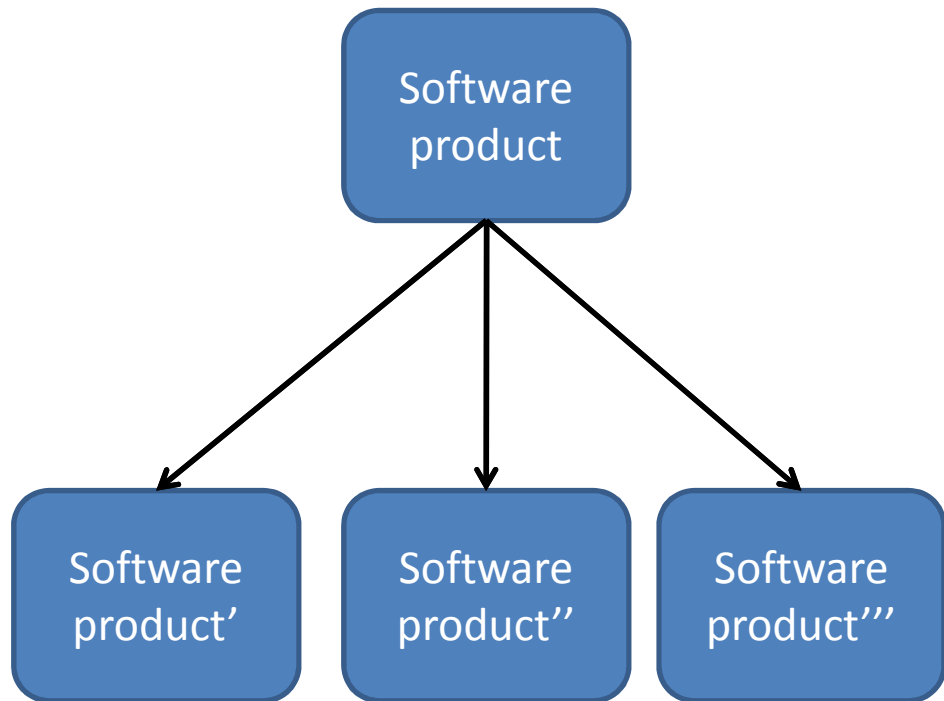
- Evolvable software products project
  - Software development group
  - IT University of Copenhagen
  - Microsoft Dynamics, DHI Denmark
- 
- Core ideas developed with A. Cisternino (University of Pisa)
  - Contributions from my colleagues @ SDG group

# Software Products

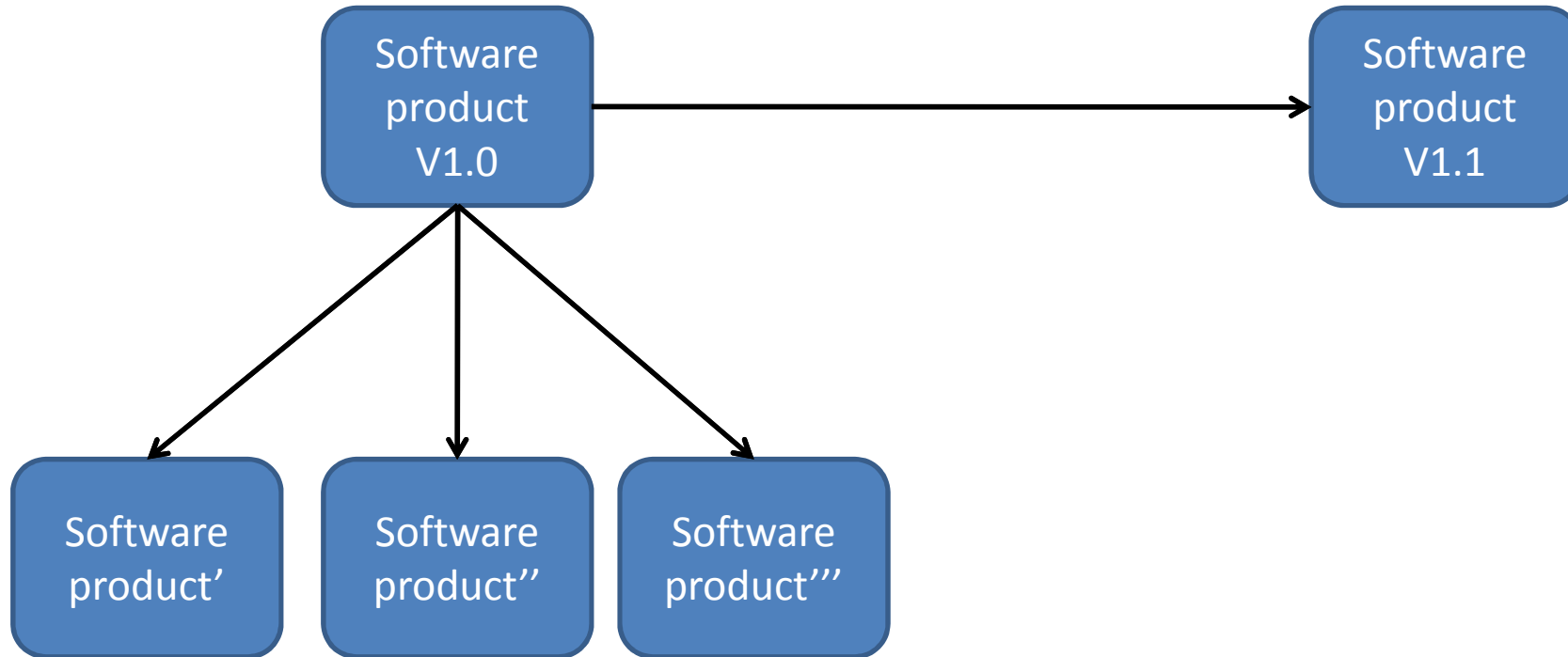


Software  
product

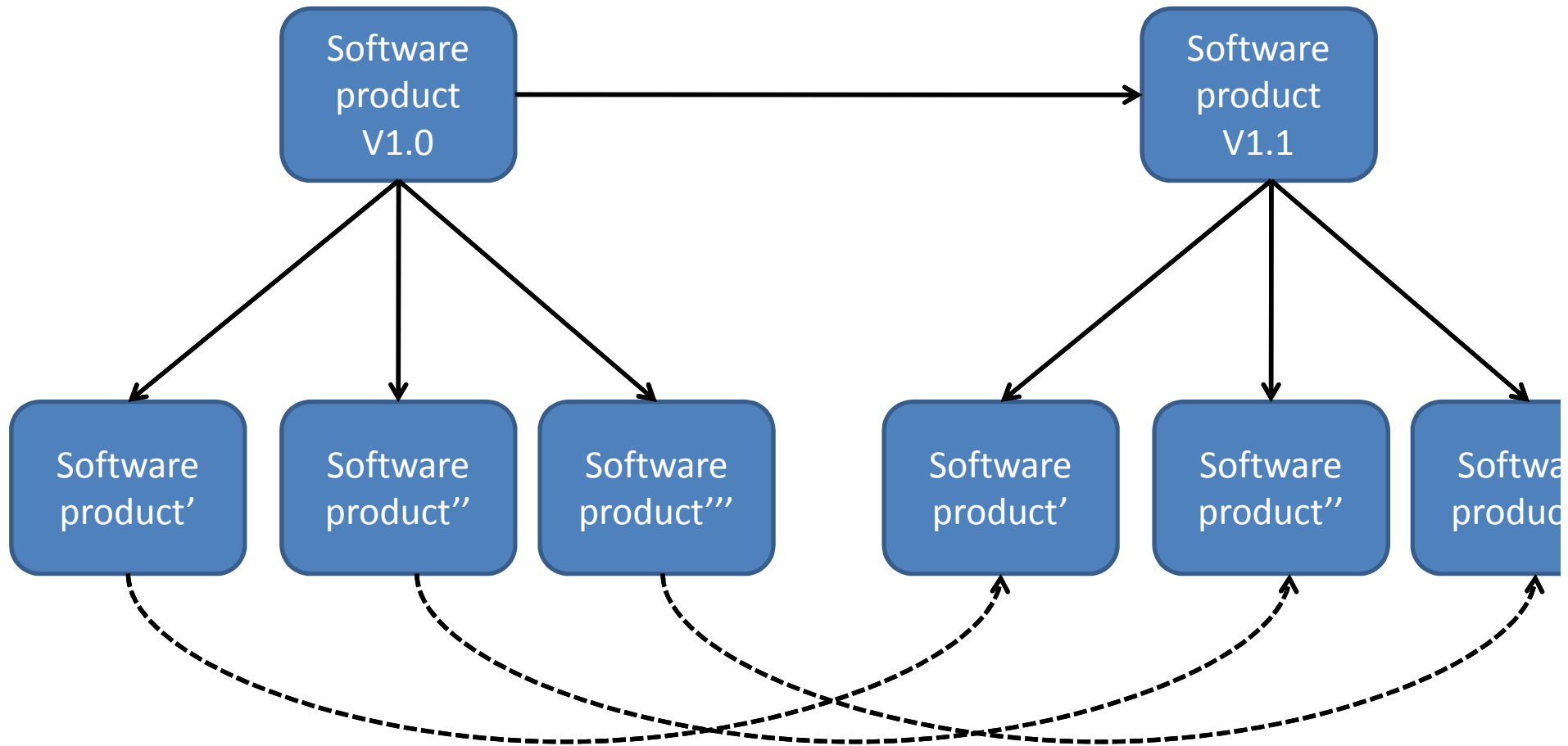
# Software Products (1<sup>st</sup> Problem)



# Evolvable Software Products



# Evolvable Software Products (2nd Problem)



# Why is it an important problem?

- Yearly spending on Enterprise Systems worldwide [Moller2004] : **US \$ 18 billion**
- An ERP system version upgrade cost: **10%-15%** of the initial investment
- *Upgrade happen approx. every 2-3 years*
- ***“8% of our customers are happy with upgrades”, Microsoft.***  
(Microsoft Dynamics employee giving a talk on upgrades at the Microsoft Convergence Conference in Munich 2006).
- "Customizations that must be carried over from one version of enterprise software to the next are the biggest **technology headache and ROI killer** that CIOs face in upgrades" [Beatty et al. 2006, Communication of the ACM].

# Empirical grounds

- **Microsoft Dynamics Enterprise Resource Planning Systems**
- **Online survey, 45 answers (Dynamics professionals, world-wide)**
- **3 Head-to-head interviews (Partners, DK)**
  
- Y.Dittrich, S. Vaucouleur.  
“Customization and Upgrade of ERP systems”.  
TR-2008-105
  
- Y.Dittrich, S. Vaucouleur.  
“Practices around Customization of Standard Systems”.  
ICSE 2008
  
- P.Sestoft, S. Vaucouleur.  
“Evolvable Software Products”,  
Springer LNCS Advances in Software Technology 2008

# Summary of empirical results

- Customizations cannot be anticipated
  - Various in kind
  - Target all location in the source code
  - Contrast with software product lines
- Partners' staff typically have modest CS education
- Time to market is more important than correctness

# Extremely simple running example

[...]

account1.Debit(x);

account2.Credit(x);

[...]

# Running example: desired customization

[...]

`account1.Debit(x * TAX);`

`account1.Debit(x);`

`account2.Credit(x);`

[...]

# Many instances

[...]

account1.Debit(x);

account2.Credit(x);

[...]

account3.Debit(y);

account4.Credit(y);

[...]

account5.Debit(z);

account6.Credit(z);

# In-place modifications

[...]

**account1.Debit(x \* TAX);**

account1.Debit(x);

account2.Credit(x);

[...]

**account3.Debit(y \* TAX);**

account3.Debit(y);

account4.Credit(y);

[...]

**account5.Debit(z \* TAX);**

account5.Debit(z);

account6.Credit(z);

# Upgrade problems

[...]

`account1.Debit(x * TAX);` // variables were renamed !!

`accountA.Debit(x);`

`accountB.Credit(x);`

[...]

`account3.Debit(y * TAX);` // method was renamed !!

`account3.DoDebit(y);`

`account4.DoCredit(y);`

[...]

`account5.Debit(z * TAX);` // code was removed !!

`Do_something(account5,account6);`

`Do_something_else(z);`

# Procedural abstraction

# Procedural abstraction

[...]

account1.Debit(x);

account2.Credit(x);

[...]

account3.Debit(y);

account4.Credit(y);

[...]

account5.Debit(z);

account6.Credit(z);

# Procedural abstraction

[...]

account1.Debit(x);

account2.Credit(x);

[...]

account3.Debit(y);

account4.Credit(y);

[...]

account5.Debit(z);

account6.Credit(z);

```
(Account a, Account b, double x)
```

```
{
```

```
    a.Debit(x);
```

```
    b.Credit(x);
```

```
}
```

# Named abstraction

[...]

account1.Debit(x);

account2.Credit(x);

[...]

account3.Debit(y);

account4.Credit(y);

[...]

account5.Debit(z);

account6.Credit(z);

```
Transaction(Account a, Account b, double x)
{
    a.Debit(x);
    b.Credit(x);
}
```

# C# version

```
public void Transaction(Account a, Account b, double x)
{
    a.Debit(x);
    b.Credit(x);
}
```

# Procedural abstraction

[...]

account1.Debit(x);

account2.Credit(x);

[...]

account3.Debit(y);

account4.Credit(y);

[...]

account5.Debit(z);

account6.Credit(z);

# Procedural abstraction

[...]

**Transaction**(account1, account2, x);

[...]

**Transaction**(account3, account4, y);

[...]

**Transaction**(account5, account6, z);

# Dynamic binding

```
public virtual void Transaction(Account a, Account b, double x)
{
    a.Debit(x);
    b.Credit(x);
}
```

```
public override void Transaction(Account a, Account b, double x)
{
    a.Debit(x * TAX);
    a.Debit(x);
    b.Credit(x);
}
```

# Dynamic binding

```
public virtual void Transaction(Account a, Account b, double x)
{
    a.Debit(x);
    b.Credit(x);
}
```

```
public override void Transaction(Account a, Account b, double x)
{
    a.Debit(x * TAX);
    a.Debit(x);
    b.Credit(x);
}
```

**Anticipation !!**

# What the textbooks say

- If A and B are likely to change together, then put them in the same module (Parnas 70's)
- Granularity of “Module” can vary (procedure, class, assembly, etc.)

# The crystal ball assumption

- If A and B are **likely** to change together, then put them in the same module
- Decomposition into modules requires anticipation of **likely** changes
- Something we cannot afford



# Event solution

[...]

Some\_code;

Raise\_transaction\_event;

Yet\_some\_other\_code;

Raise\_transaction\_event;

More\_code;

[...]

- Where to send the event ?
- What kind of event to send?

# Problem/Solution space

Solutions	Customization without anticipation	Resilience to upgrades
In-place modifications	Yes	No

# Problem/Solution space

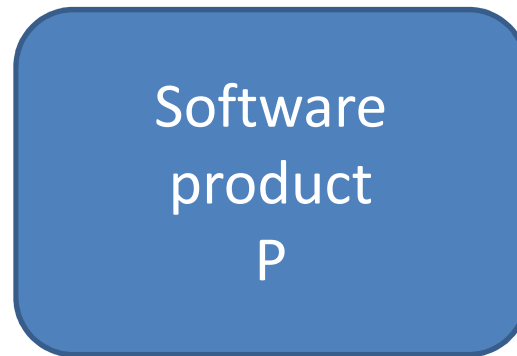
Solutions	Customization without anticipation	Resilience to upgrades
In-place modifications	Yes	No
Usual abstraction-based techn.	No	Yes

# Problem/Solution space

Solutions	Customization without anticipation	Resilience to upgrades
In-place modifications	Yes	No
Usual abstraction-based techn.	No	Yes
Our goal	Yes	Yes

- In-place modifications are **sensitive to upgrades**
- Useful abstractions requires **anticipation**

# Our approach: Glass-box



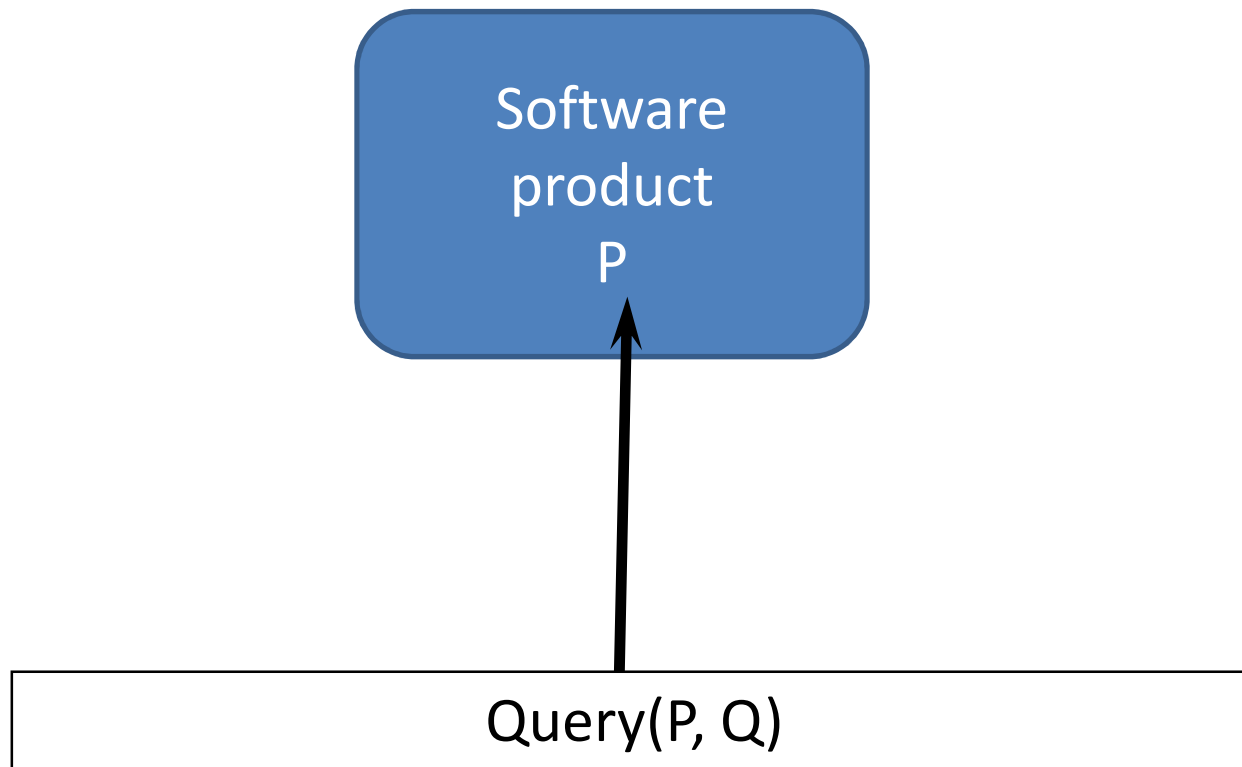
- Approx. 1 million lines of code, C#/Java like language
- No special markers
- We want to customize this product
- We want the customization to be portable to newer versions

# Partners write code queries

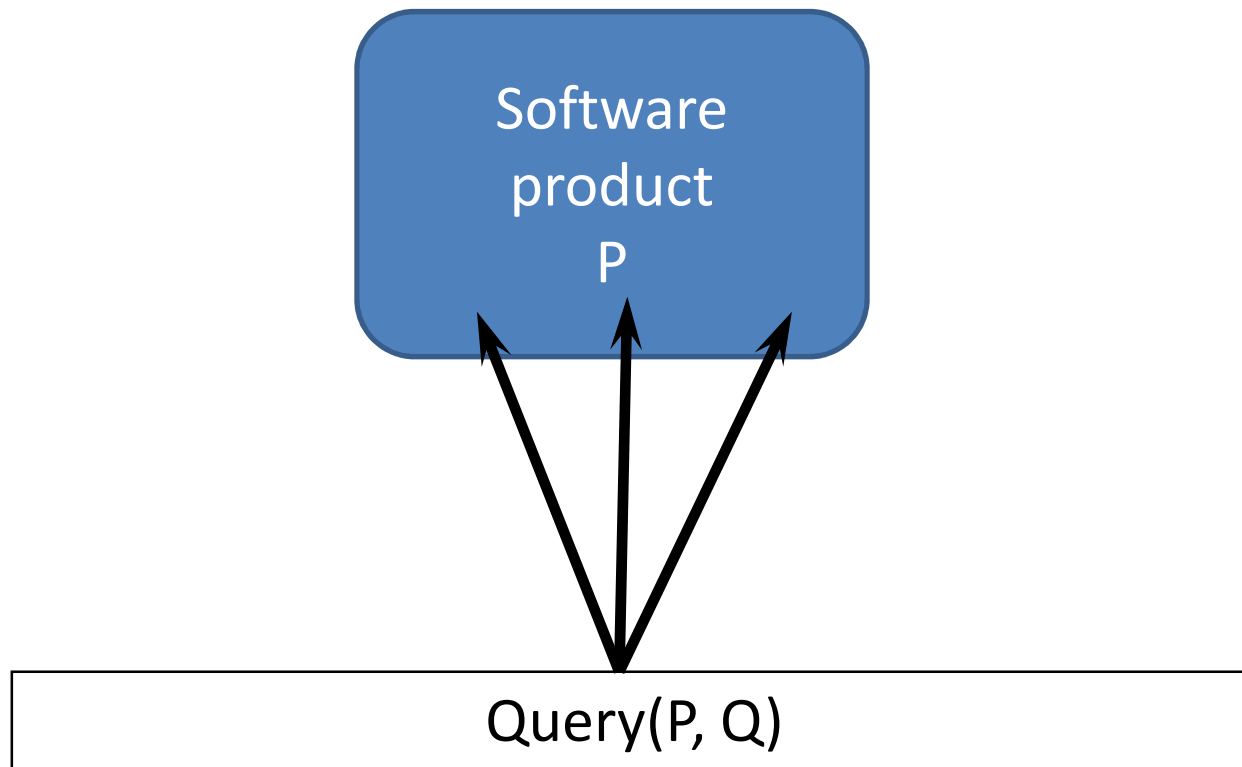
Software  
product  
P

Query(P, Q)

Result = { CodeFragment }



{ CodeFragment1, CodeFragment2, CodeFragment3... }



# Query language: what we **DON'T** want

Where: Variable(x) and Variable(y) and  
Type(x,"Account") and Type(y,"Account") and  
Statement(s1) and Statement(s2) and Method(m1)  
and Method(m2) and MethodName(m1,"Debit") and  
MethodName(m2,"Credit") and Double(x) and  
MethodCall(s1,m1) and ActualArgument(s1,x) and  
ActualArgument(s2,x) and Consecutive(s1,s2) and ...

Do: ....

# Query language: Code query by example

**Easy** for non computer science experts

a.Debit(x);

b.Credit(x);

# Query methods defined using .NET attributes

[Query]

```
public void DebitCredit(...) {  
    a.Debit(x);  
    b.Credit(x);  
}
```

- Written by partners
- Not to be executed! Used for querying..

# Typed formal parameters

[Query]

```
public void DebitCredit(Account a, Account b, double x) {  
    a.Debit(x);  
    b.Credit(x);  
}
```

# Named query method (abstraction)

[Query("Transaction")]

```
public void DebitCredit(Account a, Account b, int x) {  
    a.Debit(x);  
    b.Credit(x);  
}
```

# Query class

```
public class Partner1 {  
    [Query("Transaction")]  
    public void DebitCredit(Account a, Account b, int x) {  
        a.Debit(x);  
        b.Credit(x);  
    }  
}
```

# Existing Software Product (no markers)

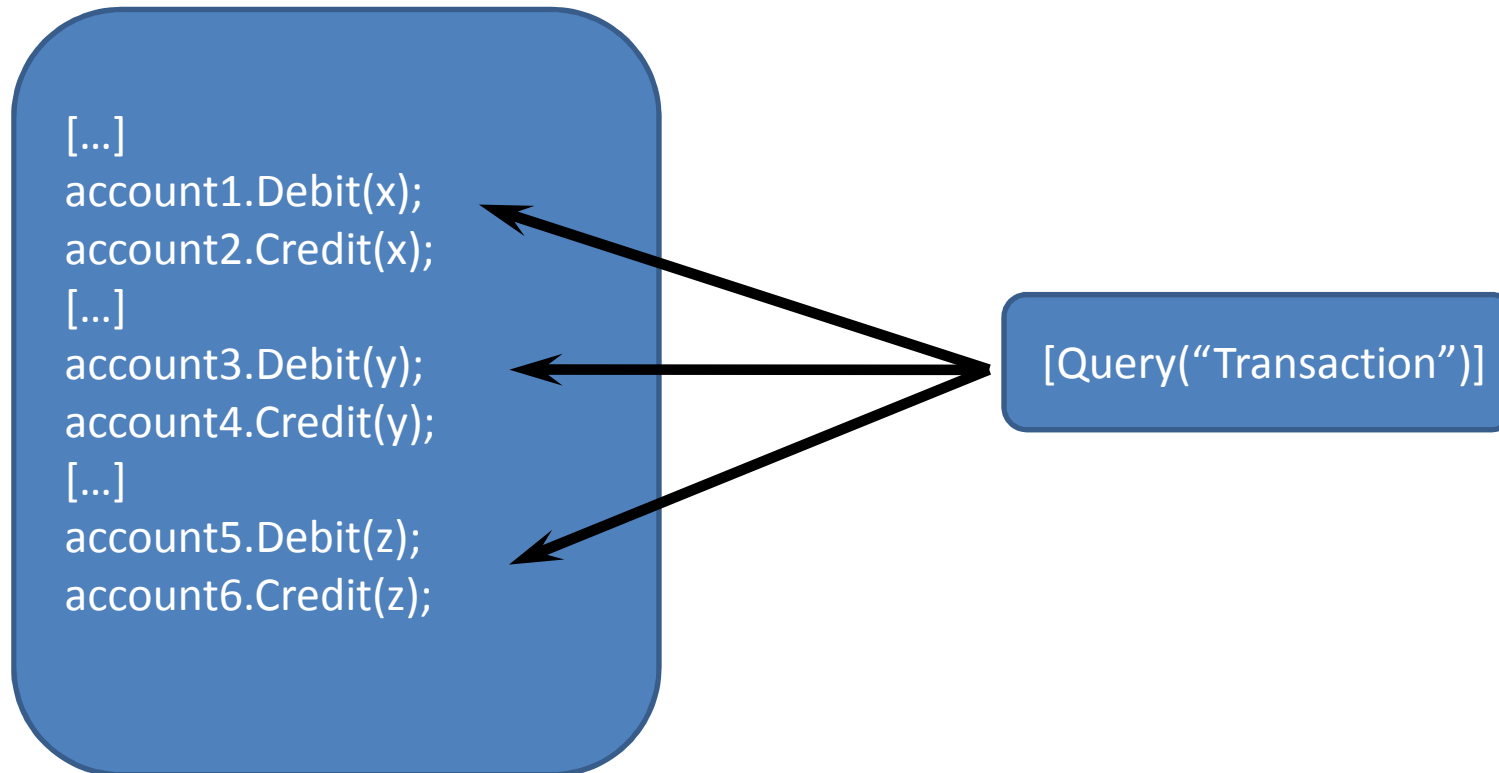
```
[...]  
account1.Debit(x);  
account2.Credit(x);  
[...]  
account3.Debit(y);  
account4.Credit(y);  
[...]  
account5.Debit(z);  
account6.Credit(z);
```

# A set of queries

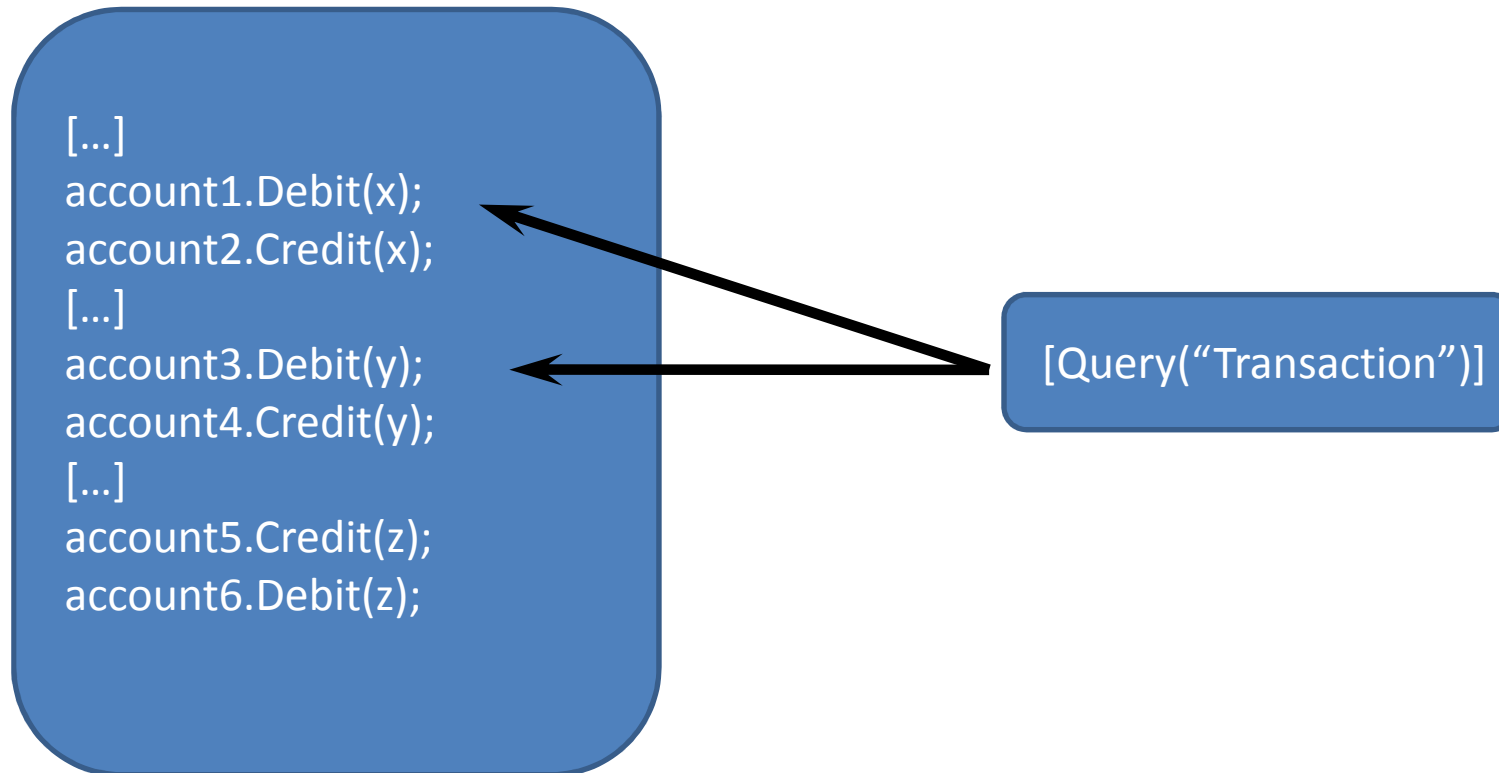
```
[...]  
account1.Debit(x);  
account2.Credit(x);  
[...]  
account3.Debit(y);  
account4.Credit(y);  
[...]  
account5.Debit(z);  
account6.Credit(z);
```

```
[Query("Transaction")]
```

Match: SoftwareProduct x Query -> CodeFragment



# What if.. Credit then Debit ?



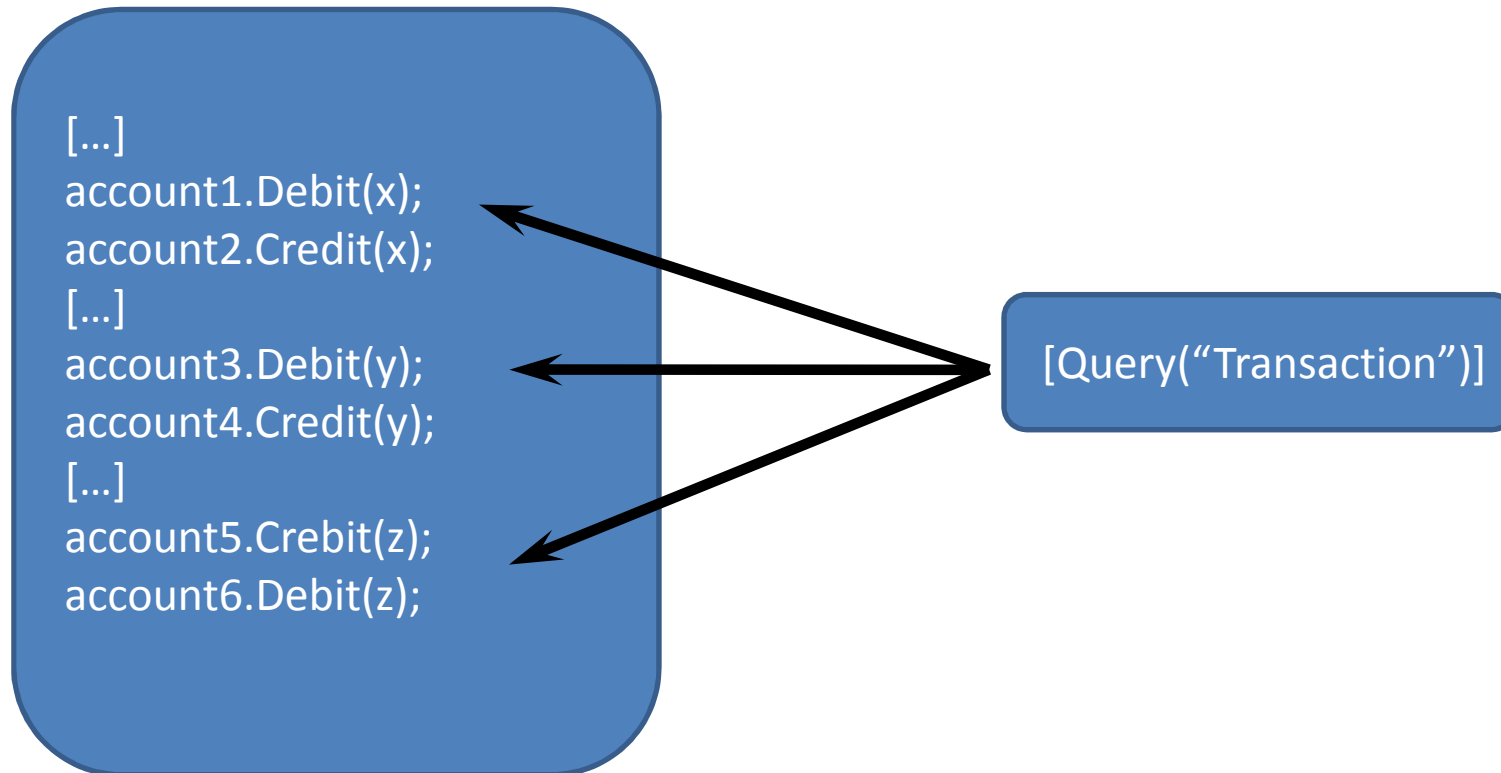
# Back to the code query

```
public class Partner1 {  
    [Query("Transaction")]  
    public void DebitCredit(Account a, Account b, double x) {  
        a.Debit(x);  
        b.Credit(x);  
    }  
}
```

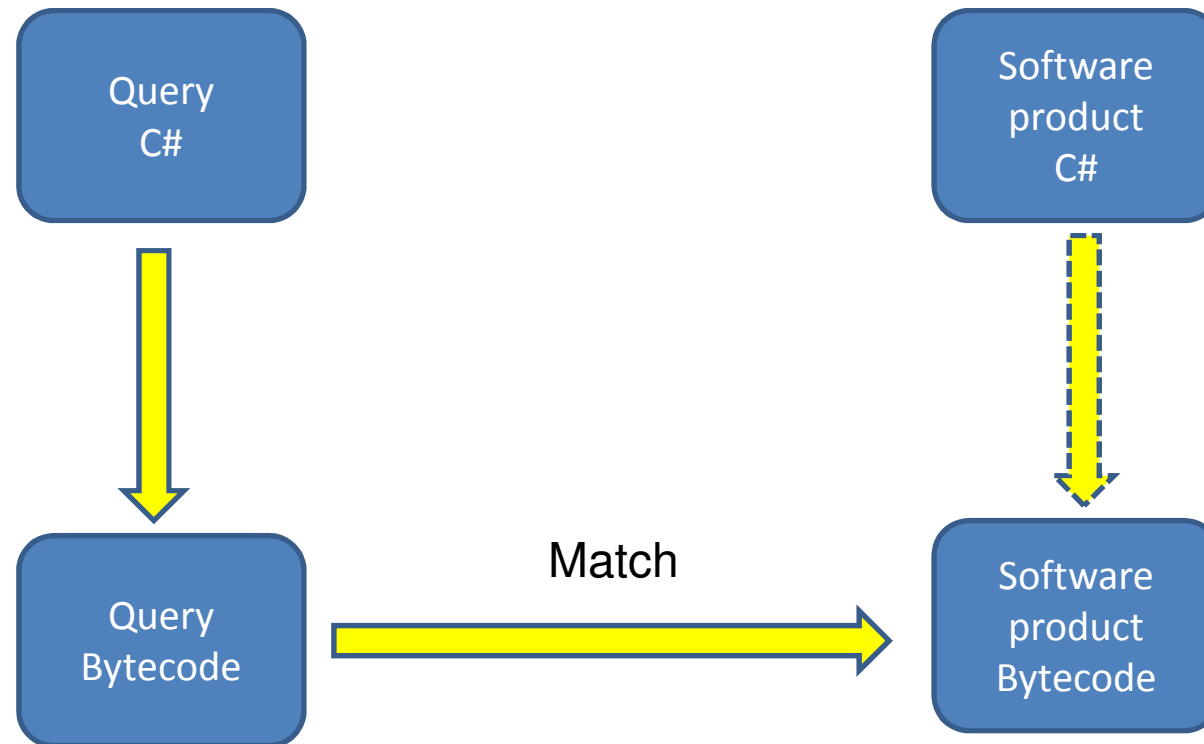
# Add a new query method

```
public class Partner1 {  
    [Query("Transaction")]  
    public void DebitCredit(Account a, Account b, double x) {  
        a.Debit(x);  
        b.Credit(x);  
    }  
    [Query("Transaction")]  
    public void CreditDebit(Account a, Account b, double x) {  
        a.Credit(x);  
        b.Debit(x);  
    }  
}
```

# Match

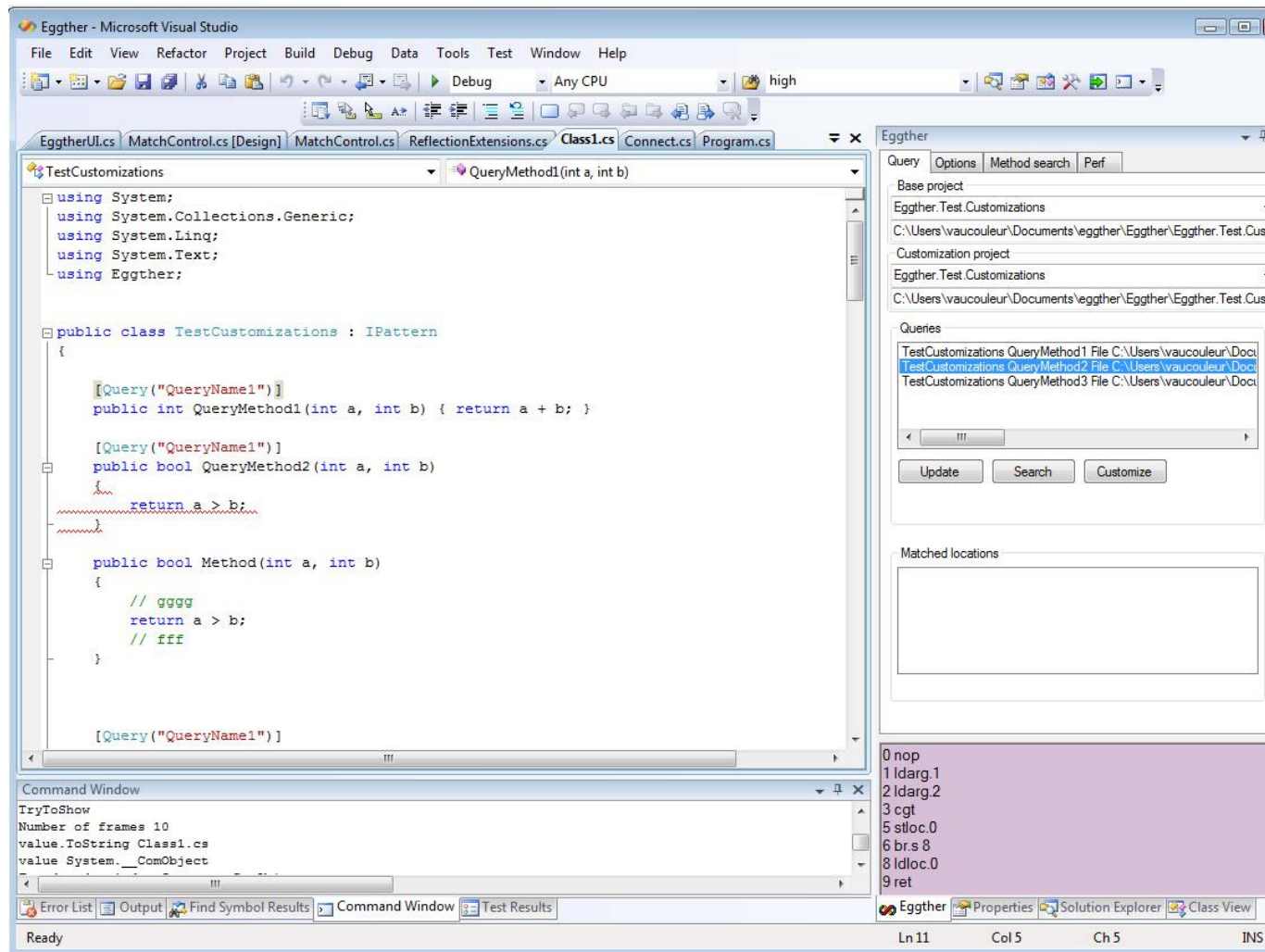


# Implementation approach: Query

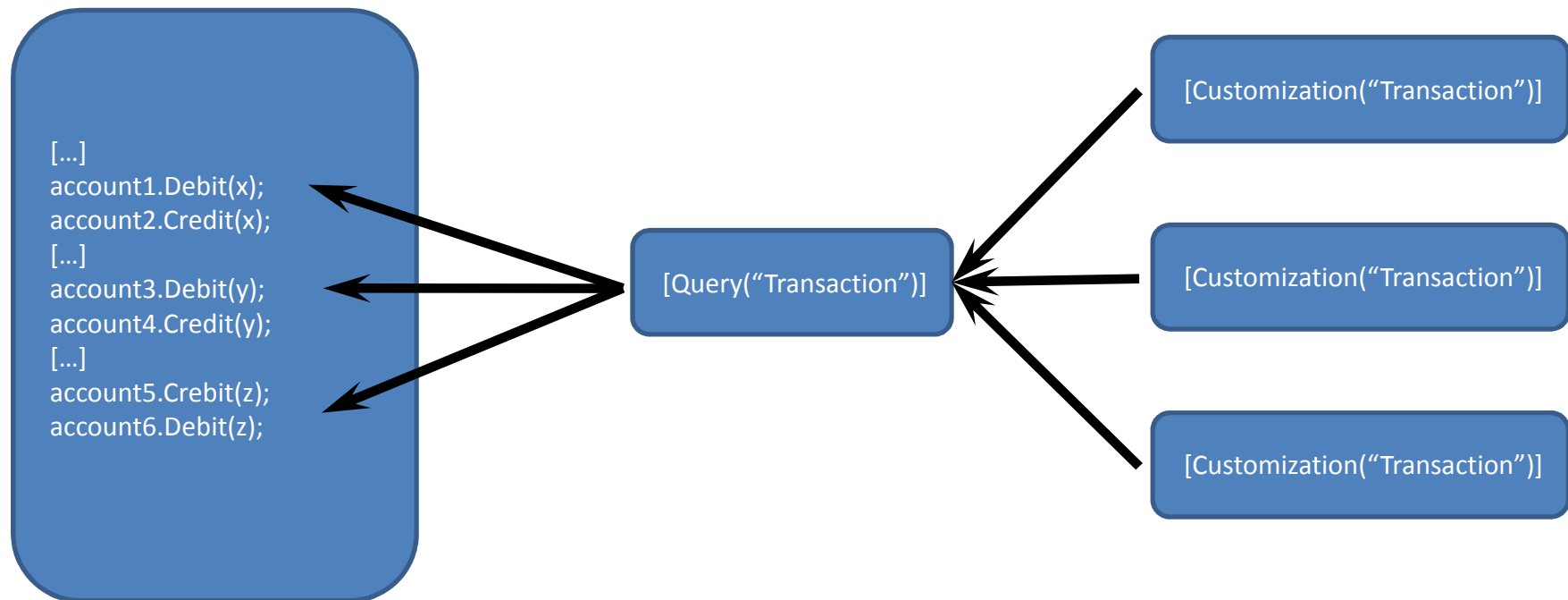


# Tool support Visual Studio plugin

- Show result of code queries (at design time)



# Part 2: Customization



# Customization

[Customization("Transaction")]

[Customization("Transaction")]

```
static public void Tax(...) {
```

```
}
```

```
public class CustomizationDenmark {  
    [Customization("Transaction")]  
    static public void Tax(...) {  
  
    }  
}
```

[Customization("Transaction")]

```
static public void Tax(...) {
```

```
}
```

```
}
```

# Binding

[Customization("Transaction")]

```
static public void Tax(Account a, Account b, double x) {  
    a.Debit(x * TAX);  
}
```

[Query("Transaction")]

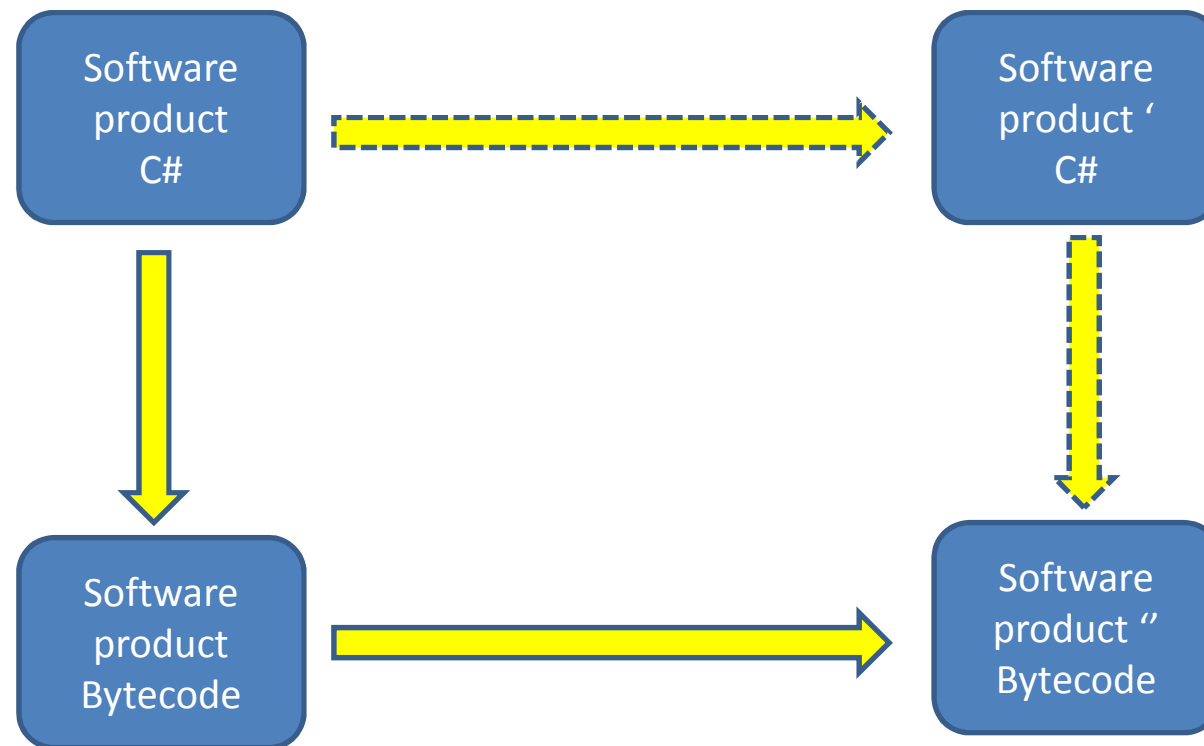
```
public void DebitCredit(Account a, Account b, double x) {  
    a.Debit(x);  
    b.Credit(x);  
}
```

```
[...]  
CustomizationDenmark.Tax(a, b, x);  
a.Debit(x);  
b.Credit(x);  
[...]
```

# Upon upgrade

- Developers re-apply code queries
  - Code query find the new locations
  - Developer inspect the results & modify the queries to fit the new version

# Implementation approach: Customization



- Pros

- Can use **standard** .NET compilers (C#,VB.NET, etc.)
  - Fast
  - Supported by Microsoft (language evolution)
- Can use **standard** IDE (Visual Studio)
  - Syntax highlighting, Refactoring tools, Debugging etc.
  - Static typing at design time within IDE
- Support for **multiple** languages

- Pros

- Can use **standard** .NET compilers (C#,VB.NET, etc.)

- Fast

- Supported by Microsoft (language evolution)

- Can use **standard** IDE (Visual Studio)

- Syntax highlighting, Refactoring tools, etc.

- Static typing at design time within IDE

- Support for **multiple** languages

- Cons

- Bytecode instrumentation can be tricky

- Non-local transformations (C# anonymous methods)

- Problem with context sensitive optimizations

Thanks, Questions ?

<http://eggther.org>