Evolvable Software Products: Technologies for Customization

Excerpts from Lipari Software Engineering Summer School Lectures (July 2007)

Project *Designing Evolvable Software Products (Microsoft Development Center Copenhagen, DHI Water and Environment, ITU)*

Peter Sestoft

IT University of Copenhagen

ESP Network lecture

Wednesday 5 December 2007

Plan

- Software products and upgrades
  - McIlroy, Parnas, Lehmann, Perry
- Collection library customization
  - Feature-oriented programming (Batory)
- Aspect-weaving for .NET
  - Layered architecture customization (AX)
- Code duplication discovery

Software products

- A software product is sold or applied multiple times; in contrast to a software project
- Example software products
  - Microsoft Office suite, Oracle database, …
  - Enterprise software: SAP, Microsoft Dynamics, …
- Software products are often highly *customizable* to the use context

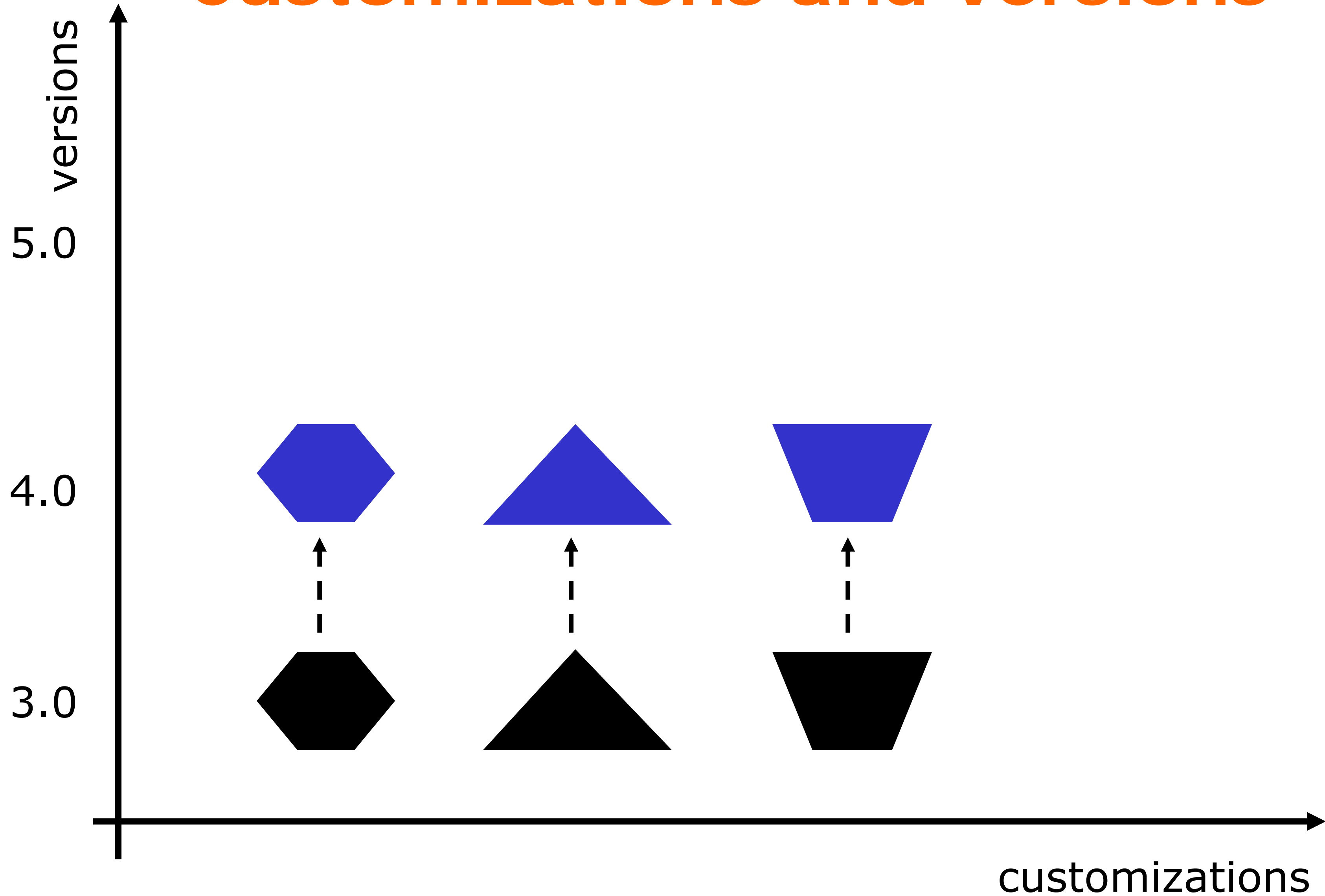
Software evolution

- Software either dies or evolves
- Witness version numbers, such as
  - Linux kernel 2.6.8-3
  - C# 2008 compiler version 3.05.20706.1
  - Oracle database version 9.2.0.6.0
- Evolution is not an accident
- Lehman 1980: S-, P-, and E-programs; Specification-, Problem-, Embedded-
- E-programs, often enterprise systems, aim to change the environment, which leads to revised software requirements

The upgrade problem

- There is (often) a conflict between customizations and evolution
- Scenario:
  - Company buys Dynamics AX 3.0
  - Company customizes it to the business
  - Dynamics AX 4.0 gets released
  - The customizations must be ported to 4.0; lots of work and cost for no obvious benefit

Customizations and versions



Case studies

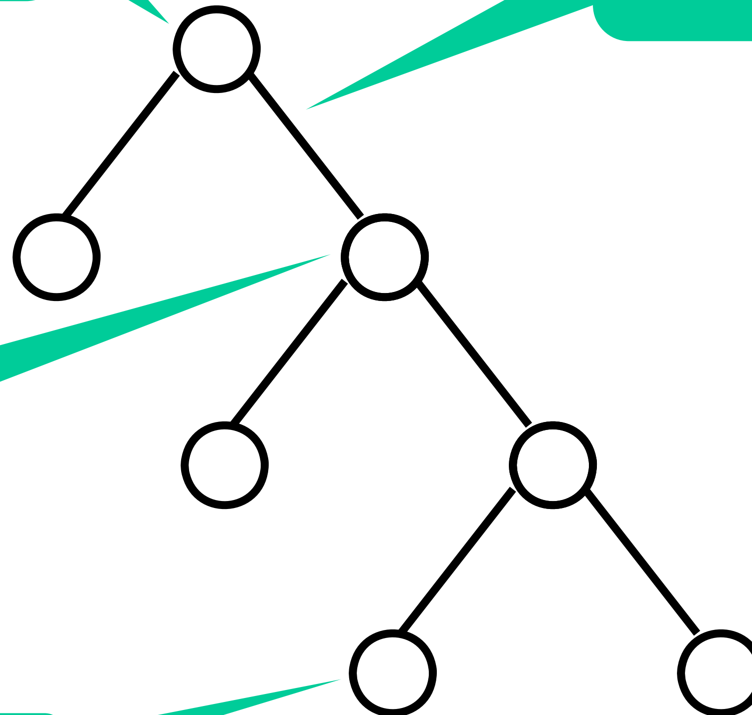
- Microsoft Dynamics AX
  - Multilayer customization
  - Microsoft/partner/user company ecosystem
  - Developed by Microsoft in the US and DK
- A collection library for C#/.NET (skip)
  - Developed at IT University of Copenhagen

- These require different approaches
  - Universe of C5 versions is known but large
  - The scope of extension of AX is unbounded

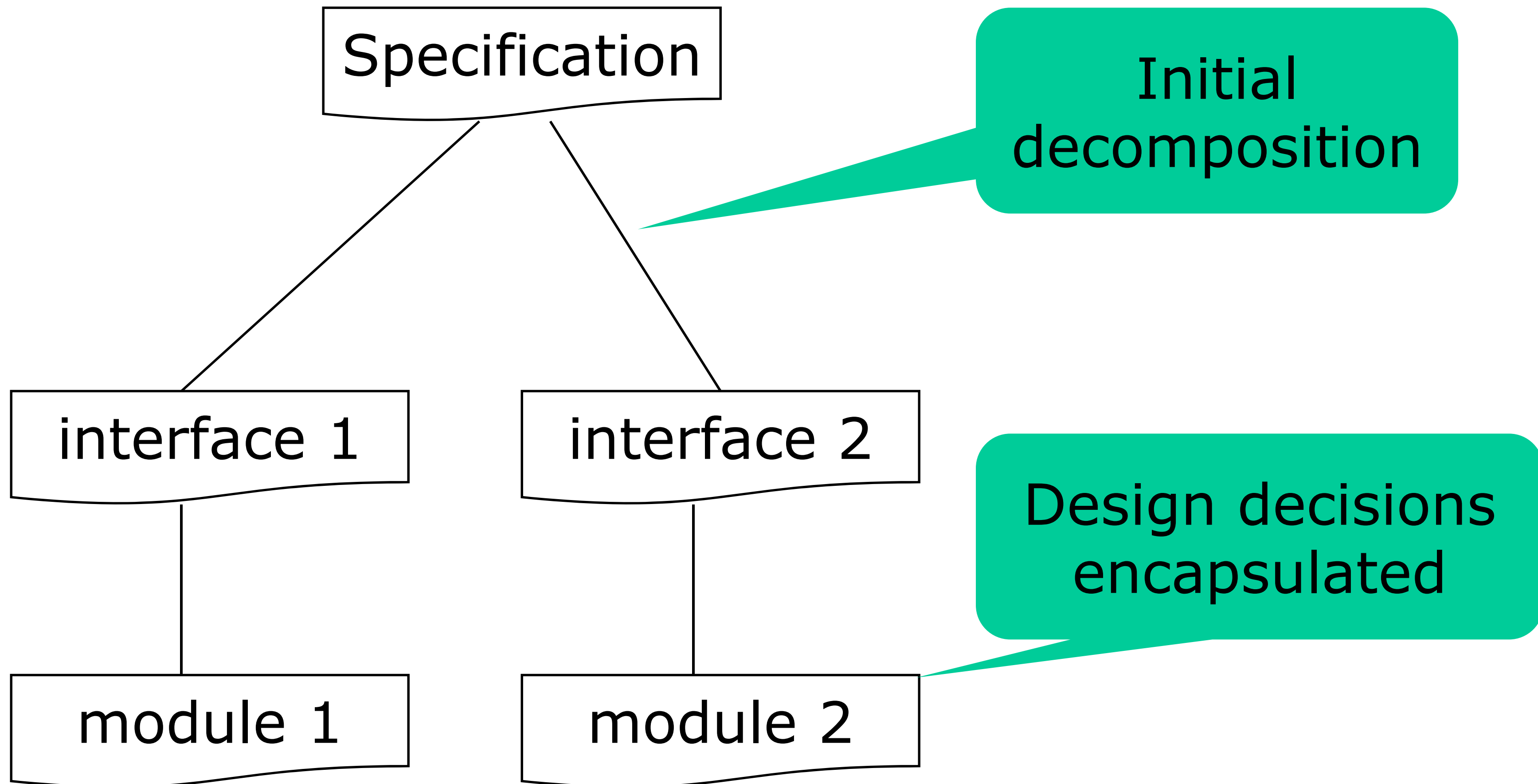
Software families (Parnas)

- Parnas 1976
  - Dijkstra 1970: Stepwise refinement; each design decision is a branching point
  - Parnas 1972: Module interfaces; each design decision is encapsulated in a module
  - A muted critique of Dijkstra …

Dijkstra stepwise refinement



Parnas module interfaces



Software families (Perry 1989)

- The InScape system 1989
  - Explicit formal module interfaces
  - Supports division of work
  - Supports **evolution of** modules and of their **interface specifications**
  - Hence takes Parnas's ideas further
- What became of InScape?
  - Developed at Bell Labs, C-oriented
  - Obsoleted by PL-development (SML, Java)?
- Lots of specification languages and tools: VDM, RAISE, B, Z, UML/OCL, JML, Spec#, Alloy, …

Evolution of specifications

- Parnas’s message:
  - A program family consists of modules with explicit interface specifications
  - Interface specifications allow module implementations to change independently
- Perry’s message:
  - The interface specifications evolve too
  - Need tool support to analyse interface change impact
  - Formal specifications enable tool support
- Most developers appreciate type checkers, null reference checks etc, but not formal specifications – why?

Microsoft Dynamics NAV and AX

- Enterprise systems – for organizations
- Layered:
  - **Microsoft** makes kernel, the base system
  - Some **partner** companies develop add-ins, specialized solutions, e.g. for
    - Pharmacies
    - Brake pad manufacturers
    - Government branches
    - …
  - A **consultant** company (or the **user company** itself) builds further customizations on top

Dynamics NAV versus AX

- Dynamics NAV (formerly Navision)
  - Many installations, smaller customizations
  - Many partner companies; often former accountants, not educated SW developers
  - Internal language C/AL fairly simple
- Dynamics AX (formerly Axapta)
  - Fewer and larger installations, more complex add-ons, larger customizations
  - Partners employ SW professionals mostly
  - Some large partners, e.g. Accenture
  - Internal language X++ similar to Java
- Ca. 1 million people use NAV or AX

Simple customization are easy to upgrade

- The story of one NAV partner
  - Small customization projects: 50-500 man hours, one or two developers
  - Manually mark modified code like this:

```
// >> 07/PS
… customized code …
// <<
```

  - Use text merge and editor when upgrading
  - Charge a fixed price (10%) of original work
  - Bear the risk and be rewarded for discipline
- Doesn't work in AX, customizations too large

Dynamics AX layers

	What	**By**
USR	User/department-specific functionality	cust
CUS	Company-wide specific functionality	cust
VAR	Business solutions, unrestricted	ptnr
BUS	Business solutions, certified add-ons	ptnr
LOS	MS-made local business solutions	MS
DIS	Critical hotfixes to SYS and GLS	MS
GLS	Country-specific functionality	MS
SYS	System, core functionality	MS

kernel

How are customizations made

- A layer contains application elements, such as table, field, form, class, method element, …
- E.g. Sales Order form; or Volume Discount method
- Higher-level (custom) element shadows lower ones by ID

USR				
CUS				
VAR				
BUS			shadowed	
LOS				
DIS				
GLS				
SYS		shadowed	shadowed	

The problem with code (method) customizations

- Now, a customization:
  - Copy method code from lower layer
  - Edit to remove, add or change functionality
- Later, a kernel application upgrade:
  - Find changed lower layer elements that had been customized
  - In each case, decide whether
    - (a) new lower layer functionality makes customization unnecessary; then remove it
    - (b) the customization continues to work; then copy it to a new customization of the lower layer code
    - (c) the customization no longer works; then design and implement a new one
  - This requires insight in both versions of AX, in the old customizations, and in the reason for them

Hence upgrade trouble

- New NAV and AX versions are released every 2 years
- Often the old system is "good enough" and upgrades are costly; up to 30% of original customization price
- So customers are not eager to upgrade
- But the systems are business critical and cannot just be discarded
- MS and partners must support old versions a long time, itself very costly…
- We hear similar things about SAP, another enterprise system…

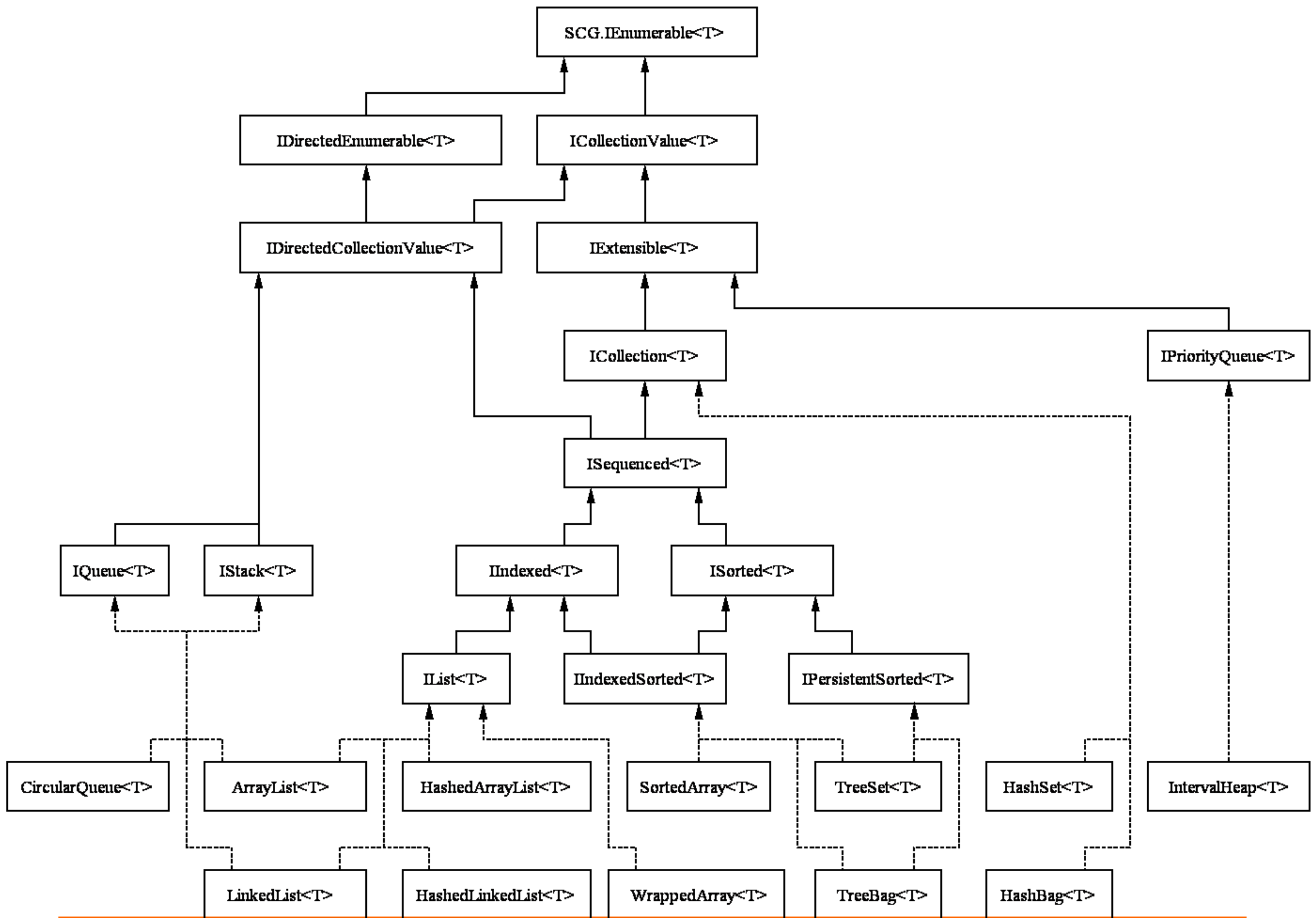
Desirable properties

- Ease of upgrade
  - Customizations should be usable on the new kernel
- Quality assurance must ensure that
  - Customizations are compatible with the new kernel
  - No two customizations conflict
  - Preferably checked statically, at build time
- Some tools for quality assurance
  - Regression tests of the upgraded system; but not used systematically by partners
  - Static checking; necessarily conservative but much better than runtime failures
- Familiarity
  - Should preserve the layered model, and ecosystem

The C5 Generic Collection Library for C# and .NET

- Array lists, linked list, sets, bags, stacks, queues, priority queues etc in coherent design
- All functionality described by interfaces; “code to interface not implementation”
- Rich functionality, including
  - Hash-indexed array lists and linked lists
  - Slidable updatable sublist views
  - Snapshots of tree-based sets
  - Event listeners on all updates
- 27 000 lines of C#, 28 000 lines unit tests

See http://www.itu.dk/research/c5/


SCG.IEnumerable<T>
IDirectedEnumerable<T>
ICollectionValue<T>
IDirectedCollectionValue<T>
IExtensible<T>
ICollection<T>
IPriorityQueue<T>
ISequenced<T>
IQueue<T>
IStack<T>
IIndexed<T>
ISorted<T>
IList<T>
IIndexedSorted<T>
IPersistentSorted<T>
CircularQueue<T>
ArrayList<T>
HashedArrayList<T>
SortedArray<T>
TreeSet<T>
HashSet<T>
IntervalHeap<T>
LinkedList<T>
HashedLinkedList<T>
WrappedArray<T>
TreeBag<T>
HashBag<T>

C5 library adoption

- Version 1.0.0 released 30 January 2006
- Documented in 250 page tech report
- Ca. 8000 downloads (order of magnitude)
- Included in Mono’s .NET implementation
- Positive feedback: “really cool”, “very nice”, “excellent, very well thought out”, “amazing”, “phenomenally useful”, “wonderful library”, ...
- Nine bugs fixed until 1.0.2 in June 2007
- Users in universities, engineering, finance, banking, defence, ...
- Article in Dr Dobb’s journal July 2007

BUT … implementation is complex

- Rich functionality costs **space** for extra fields
- A LinkedList<T> has many special fields:
  - c for Count;
  - e for events;
  - f for enumeration;
  - h for caching hash codes;
  - v for slidable views
- HashedLinkedList<T> has even more fields …
- Rich functionality costs **time** to maintain and test fields

e	`EventBlock<T> events`
	`bool isSortedType`
	`bool isReadOnlyType`
	`IEqualityComparer<T> …`
h	`int iUnsequencedHashCode`
h	`int iUnsequenceHashCodeStamp`
c	`int size`
f	`int stamp`
h	`Int iSequencedHashCode`
h	`Int iSequencedHashCodeStamp`
	`Node startsentinel`
	`Node endsentinel`
	`bool fifo`
v	`bool isValid`
v	`int offset`
v	`List<T> underlying`
v	`WeakViewList<...> views`
v	`WeakViewList<...>.Node myWeakReference`

Example LinkedList<T> code

```
public virtual T Remove() {
  updatecheck();
  if (size == 0)
    throw new NoSuchItemException("List is empty");
  T item = fIFO ? remove(startsentinel.next, 0) : remove(endsentinel.pre
  (underlying ?? this).raiseForRemove(item);
  return item;
}

T remove(Node node, int index) {
  fixViewsBeforeSingleRemove(node, Offset + index);
  node.prev.next = node.next;
  node.next.prev = node.prev;
  size--;
  if (underlying != null)
    underlying.size--;
  return node.item;
}
```

For iteration and hashcode caching

For events

For sublist views

For size

For size *and* sublist views

Still, performance is good

- Comparison of red/black tree set collections (Goletas, June 2006)

Operation	.NET	C5	PowerCollections
Add	43.386	48.683	52.823
ContainsKey	21.474	23.398	22.602
foreach	2.0734	2.176	(Out of RAM)
Remove	37.801	31.485	46.540
Add ascending	19.651	21.237	22.714
Remove desc.	11.998	9.527	18.560

Wish: A practical specialization tool

- A means to specify specialized classes, eg: “*Make me a linked list class*
  - *with hash index*
  - *with constant-time Count property, but*
  - *no slidable views,*
  - *no event listeners,*
  - *no support for enumeration, and*
  - *no caching of the list's hashcode*".
- A tool that
  - Checks the consistency of these choices
  - Generates an implementation that
    - carries no superfluous data and
    - performs no superfluous runtime actions.
  - Possibly generates specialized (reduced) test cases

Lots of prior work

- McIlroy (1968, dreaming): "*Use a String Associates A4 symbol table, in size 500x8*"
- Batory et al. 1985-1990: Genesis, building blocks for database systems
- Batory, O'Malley 1993: Genvoca
- Batory et al. 1992: Scalable Software Libraries, generator of data structure implementations
- Smaragdakis, Batory 1997: DiSTiL, generator of data structures implemented as Intentional Programming library
- We might/should build on this and later work, but …
  - Unlikely to find good algorithms automatically, e.g. for hash-indexed linked lists with slidable views
  - Our goal is different: want to specialize, not compose
  - Probably not the right approach for enterprise systems

Universe of systems

- Generators and specializers work when the universe of systems is (large but) well-understood and relatively stable
- It holds for collection classes
- It does not hold for enterprise systems
  - New business methods are invented daily
  - New legislation is adopted daily

Customization technologies
A tangle of concepts

- Generative programming
- Feature-oriented programming
- Metaprogramming, two-level languages
- Domain-specific languages
- Model-driven development
- Frameworks
- Software product lines
- Roles in OOP (Kristensen & Østerbye)
- Interpretation vs compilation
- Partial evaluation
- Program slicing

Generative programming

- Program generators are old: a compiler is a generator of low-level programs
- But usually refers to generation in a high-level language
- McIlroy 1968 and Parnas 1976 both mention program generation
- "*Generators reduce maintenance costs, produce more evolvable software*, …" (Batory)
  - Because code is generated from something higher-level, less redundant, more maintainable
  - But, maintenance and evolution of the generator

- See also Czarnecki, Eisenecker 2000

Generative programming, standards and costs

- McIlroy in 1968 imagined these sine routine generator variation points:
  - Number of bits precision; approximating function
  - Floating- or fixed-point computation
  - Argument ranges $[0,\pi/2]$ or $[0,2\pi]$ or …
  - Robustness and failure modes
- We don't need such generators today, why?
- Hardware standardized on IEEE-754 arithmetics in 1985 (W. Kahan)
- Even if 87-bit sine would be optimal for **your** needs (astronomy, say), you cannot afford it!
- Standardization reduces choice, but also lowers cost; so you stop wanting choice …

Metaprogramming

- Reflection/reification: A program may
  - inspect its own state (what types exists…)
  - modify its own state (add types, code, …)
- Examples:
  - Smalltalk, Java & C# runtimes, classloader
  - Prolog assert/retract
- Two-level languages
  - Lisp/Scheme quasiquotation and eval
  - C# S.R.Emit, C# 3.0 expression trees
- Basically, just embedded generative programming…

Domain-specific languages

- "*Generators are compilers for domain-specific languages*" (Smaragdakis)
- Corollary: A generator defines the semantics of some domain-specific language
- Domain-specific language implementation:

	Interpreted	**Compiled**
Embedded in host language	Overloaded operators, e.g. Lava in Haskell; or reflection	C++ template meta programming (Veldhuizen)
Separate	Standard interpreter	Program generation

Frameworks

- Use standard object-oriented concepts
- Patterns of use, e.g. action listeners
- Typically with configurable objects, and with hooks/callbacks for extensible behaviour

- Examples:
  - Microsoft System.Windows.Forms and Java AWT/Swing user interface classes; action listeners react to user gestures; layout managers …
  - Java Enterprise Entity Beans, all these home/local/remote interfaces…
  - ADO.NET object-relational mapping
- Frameworks are often tiresome to use, and so get supplemented with program generators

Events/hooks/notifications

- Idea: Leave “holes” in the standard system, in the form of event calls
- When a hole is reached the event is raised (aka. observer notified, trigger triggered)
- A form of parametrization, but dynamic
- Drawbacks:
  - Possible to cause infinite event cycles
  - What can the event handler assume about the base system, and vice versa?
  - Static consistency checking difficult
  - Difficult to foresee where to put “holes”
- More static alternatives:
  - Aspects, see later
  - C# 3.0 partial methods – called only if body exists

C# 3.0 extension methods

- Add instance method to existing class or interface
- Can organize crosscutting functionality
- Eg. serialization can be added *post hoc* and in one place
- Useful if the base classes are machine-generated (from XML or DB schemas)
- But
  - poor encapsulation
  - dispatch on compile-time type, not virtual dispatch

Reflective plugin discovery

- Query assembly to find classes implementing interface IPlugin

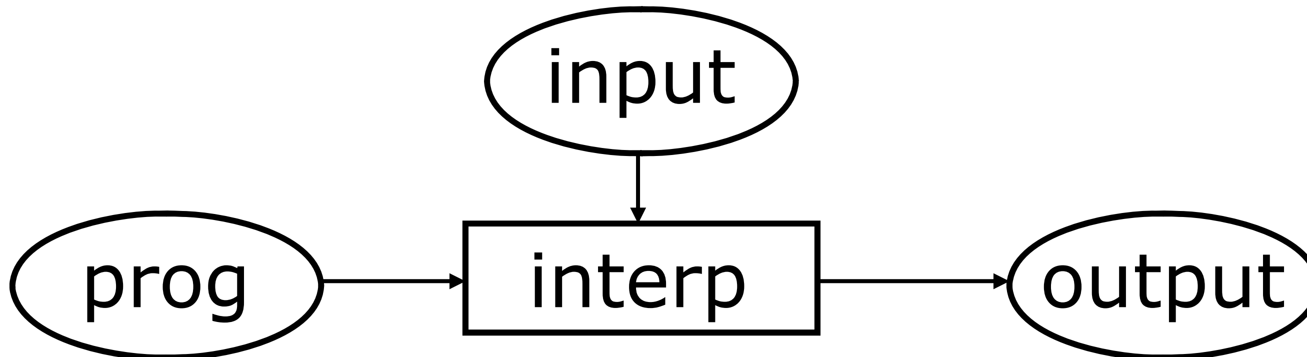
```
Assembly assembly = …
Type iPluginType = new Type("IPlugin");
foreach (Type t in assembly.GetTypes()) {
  if (t.IsClass) {
  Type[] interfaces = t.FindInterfaces(delegate(Type intf) {
    return intf == iPluginType;
  });
  if (interfaces.Length > 0)
}
```

Model-driven development

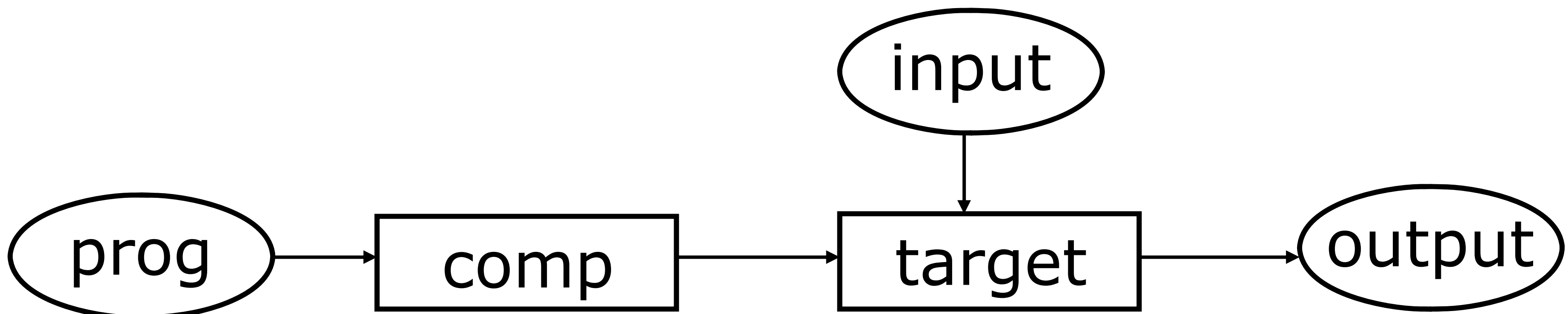
- A model is a structure adhering to a metamodel: classes, constraints, …
- The metamodel can be thought of as a domain-specific language
- (Sometimes with a vague semantics)
- A model is a program in that language
- A model (program) may be *interpreted* in a reflective architecture, or *compiled* to something else (Java, C#)

Interpreter versus compiler

Interpretation, one-stage execution:

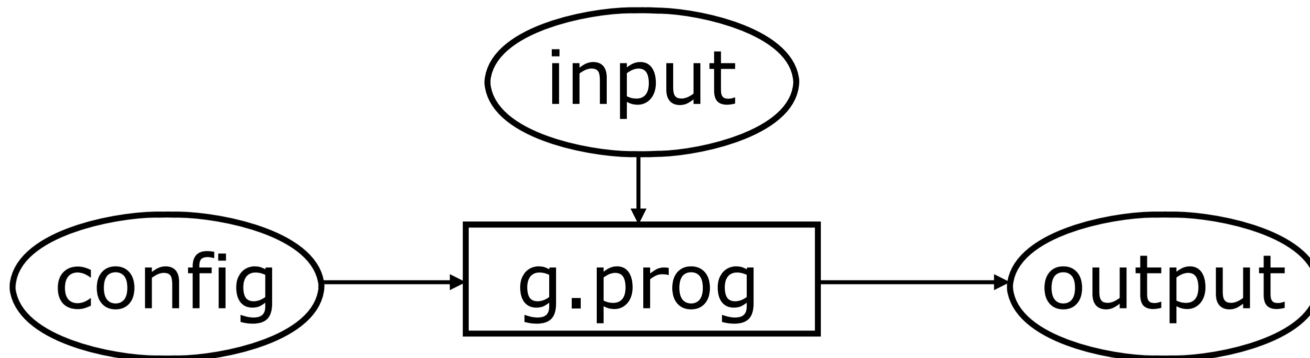


Compilation, two-stage execution:

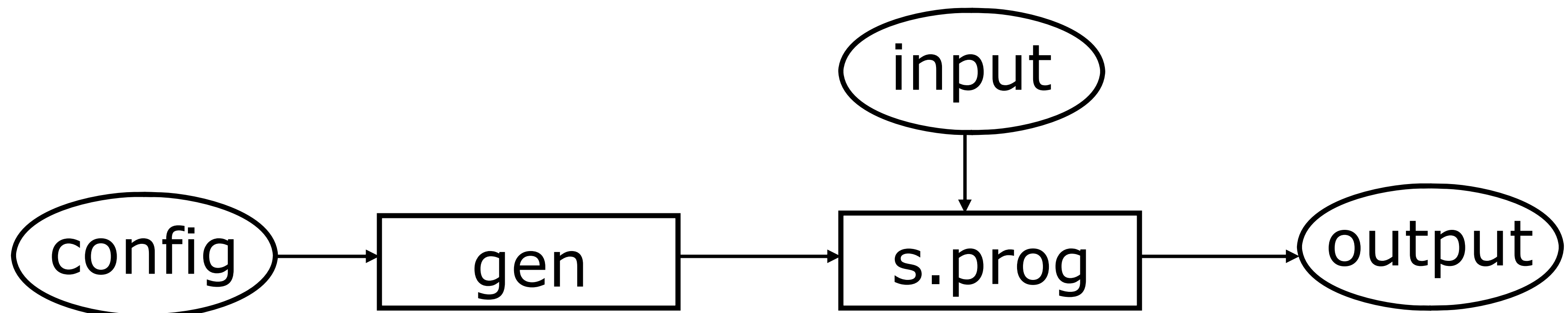


Interpretive versus generative specialization

Running a configurable general program:



Generating a specialized program:



Interpretive versus generative

- Advantages of one-stage, interpretive solution:
  - Dynamic reconfiguration without rebuild or restart
  - Avoids disrupting business (but data upgrades usually needed anyway)
- Advantages of two-stage, generative solution:
  - Early checks of configuration
  - More efficient specialized program/library
  - Small specialized program/library

Parnas on program generators

"*System generators would be completely unnecessary if we wished to build a program which at runtime could `simulate' any member of the family. Such a program would be relatively inefficient. By removing much of this variability at the time the program is generated, increases in productive capacity are made possible.*" (Parnas 1976)

- So generators are used for speed
- But speed and space (and also early checking) are often very important

Software evolution and the expression problem

"Since software evolves over time, it is essential for software systems to be extensible.

But the development of extensible software poses many design and implementation problems, especially if extensions cannot be anticipated.

*The **expression problem** is probably the most fundamental one among these problems."*

(Zenger and Odersky 2005)

History: First discussed by William Cook 1990, but named by Philip Wadler 1998

The expression problem

- Consider an expression language
  - with expressions: 117, 117+42, -(117+42)
  - with operations: eval, show, double, …
- More generally, consider software
  - with data variants: Num, Plus, Neg, …
  - with operations: eval, show, double, …
  - and each operation handles each variant differently
- How implement so as to get
  - Extensibility in both dimensions: variants,operations
  - Strong static type safety; should prevent application of an operation to a data variant it cannot handle
  - No modification or duplication of code
  - Separate compilation of variants and operations

Functional non-solution (Standard ML)

```
datatype exp =
    Num of int
  | Plus of exp * exp
fun eval e =
  case e of
      Num i => i
    | Plus(e1, e2) => eval e1 + eval e2
fun show e =
  case e of
      Num i => Int.toString i
    | Plus(e1, e2) => show e1 ^ "+" ^ show e2
```

- Adding an operation can be done in one place
- Adding a data variant requires multiple edits

Object-oriented non-solution (C#)

```
abstract class Exp {
  public abstract int Eval();
  public abstract String Show();
}
class Num : Exp {
  int i;
  public Num(int i) { this.i = i; }
  public override int Eval() { return i; }
  public override String Show() { return i.ToString(); }
}
class Plus : Exp {
  Exp e1, e2;
  public Plus(Exp e1, Exp e2) { this.e1 = e1; this.e2 = e2; }
  public override int Eval() { return e1.Eval() + e2.Eval(); }
  public override String Show() { return e1.Show() + "+" + e2.Show(); }
}
```

- Adding a data variant can be done in one place
- Adding an operation requires multiple edits (each class)

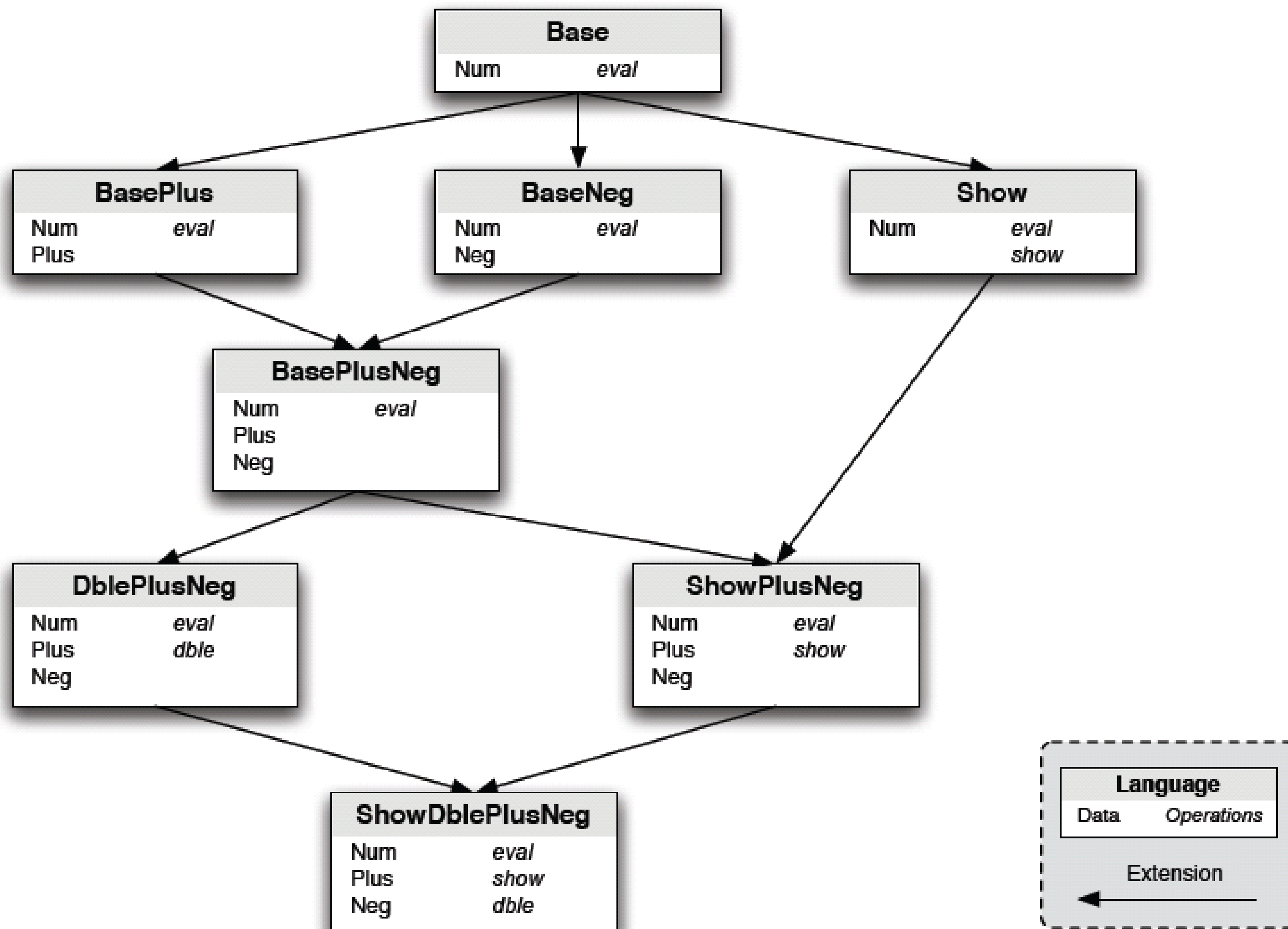
Visitor pattern non-solution

- The visitor pattern requires each data variant, or subclass, to have an **`accept(Visitor)`** method
- The operations are implemented as **`Visitor`** objects
- Adding an operation can be done in one place – add a new visitor
- Adding a data variant requires multiple edits – one to each visitor
- This makes a functional-style non-solution in an object-oriented language

Other non-solutions

- Using default methods (as in Smalltalk or Ruby)
- Runtime class checks and casts
- …

Plan for expression problem example



From Zenger and Odersky 2005

Relation to enterprise systems?

- What relation between an enterprise system and the expression problem?
- Business objects and operations
  - can be implemented and typechecked independently
  - and their composition typechecked too
- But still requires foresight:
  - How to know that the **`exp <: Exp`** technique would become useful later?
- Hence still may require restructuring
  - But at least the type system will tell us so

Aspect weaving for .NET

- Yiihaw aspect weaver (March 2007)
- MSc students Johansen & Spangenberg
- Zero-overhead aspects for .NET
- Transforms .NET assemblies
  - metadata and bytecode
  - using the Mono.Cecil library
- Allows weaving of already woven assemblies, like AspectJ for Java
- Some existing aspect weavers for .NET: AspectDNG, Aspect.NET, Rapier LOOM, NKalore, DotSpect, PostSharp, Aspect#

Core features of Yiihaw

- Input to Yiihaw
  - A *target* assembly (.exe or .dll) to add advice to
  - An *advice* assembly (.dll)
  - A *pointcut* file (.txt)
- Output from Yiihaw
  - A new woven assembly (.exe or .dll):

```
woven = weave(target, advice, pointcuts)
```

- Advice kinds:
  - **`around`**: wrap code around method bodies
  - **`insert`**: add member to class, or type to assembly
  - **`modify`**: add base class or interface to class
- Hence only static advice, no cflow
- But also no runtime or space overhead

Example aspect weaving

Pointcut file

```
insert field public instance
   int AspectConstructs:count into TargetClass;
around * instance * TargetClass:*()
   do AspectConstructs:CountAspect;
```

Target

```
.method void Method1() cil managed {
   // Code size       13 (0xd)
   .maxstack  8
   IL_0000:  nop
   IL_0001:  ldstr       "Method1"
   IL_0006:  call        WriteLine(string)
   IL_000b:  nop
   IL_000c:  ret
}
```

Advice

```
 .field public int32 count
 .method !!T CountAspect<T>() cil managed
 {
   // Code size       20 (0x14)
   .maxstack  8
   IL_0000:  ldarg.0
   IL_0001:  dup
   IL_0002:  ldfld       int32 count
   IL_0007:  ldc.i4.1
   IL_0008:  add
   IL_0009:  stfld       int32 count
   IL_000e:  call        !!0 Proceed<!!0>()
   IL_0013:  ret
 }
```

weave

Woven output

```
.field public int32 count
.method void Method1() {
  // Code size       26 (0x1a)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  dup
  IL_0002:  ldfld       int32 count
  IL_0007:  ldc.i4.1
  IL_0008:  add
  IL_0009:  stfld       int32 count
  IL_000e:  ldstr       "Method1"
  IL_0013:  call        WriteLine(string)
  IL_0018:  nop
  IL_0019:  ret
}
```

Commands for weaving

```
csc Target.cs
csc /r:..\YIIHAW.API.dll /t:library Advice.cs
..\yiihaw pointcut.txt Target.exe Advice.dll
```

- The commands:
  - Compile advice, producing **`Target.exe`**
  - Compile target, producing **`Advice.dll`**
  - Weave advice and target according to **`pointcut.txt`**, producing new **`Target.exe`**
- Compiling of advice and target checks types
- Weaver checks consistency of advice, target and pointcut file
- May run peverify to check result of weaving

Yiihaw "API" for advice methods

- **`R Proceed<R>()`** executes the target method's body and returns its result
- **`T GetTarget<T>()`** returns the target method's receiver (if instance method)

- Compiling the advice against the API gives static, pre-weaving type checking
- During weaving
  - **`Proceed`** is replaced with target's body
  - **`GetTarget`** is replaced with **`ldarg.0`**
- No overhead in the woven code!

Strongly typed advice (new)

Advice method	Some legal targets

void AdviceMethod1() {
… Proceed<Void>() …
}
	void M() { … }
	void M(bool x) { … }
	void M(bool x, int y) { … }

void AdviceMethod2(bool x) {
… Proceed<Void>() …
}
	void M(bool x) { … }
	void M(bool x, int y) { … }

int AdviceMethod3() {
… Proceed<int>() …
}
	int M() { … return … }
	int M(bool x) { … return … }

R AdviceMethod4<R>() {
… Proceed<R>() …
}
	void M() { … }
	int M() { return … }
	String[] M() { return … }

R AdviceMethod5<R>()
where R : IComparable<R> {
… Proceed<R>().CompareTo(…)
}
	int M() { return … }
	String M() { return … }
	City M() { return … }
	when City : IComparable<City>

Typed aspects eliminate need for boxing and unboxing

- In most weavers, **`Proceed()`** returns **`Object`**, so result must be tested and unwrapped:

```
int AdviceMethod() {
   Object res = Proceed();
   int i = (int)res;
   ...
}
```


- And primitive type results must be wrapped:

```
int TargetMethod() {
   int res = ... computation ...;
   return res;
}
```


- Wrapping/unwrapping gives runtime overhead
- Yiihaw avoids this entirely

Yiihaw for a layered architecture

- Yiihaw can weave advice into the result of a previous weaving
- A customization of the System layer at the Business layer can be further customized at the Customer layer:
  - (1) custBUS = BUS + weave(SYS, adv1)
  - (2) custCUS = CUS + weave(custBUS,adv2)

Example Invoice class

```
public class Invoice {
  public virtual decimal GrandTotal() {
    decimal total = ... complex computation ... ;
    return total;
  }
}
```

- Assume this is at the System layer
- We now want to give 5% discount on invoices over 10 000 Euro
- So weave Invoice at the Business layer

Adding the discount advice

```
public class MyInvoiceAspect {
  public decimal DoDiscountAspect() {
    decimal total = JoinPointContext.Proceed<decimal>();
    return total * (total < 10000 ? 1.0M : 0.95M);
  }
}
```

The pointcut file:

```
around * * decimal Invoice:GrandTotal()
  do MyInvoiceAspect:DoDiscountAspect;
```

The resulting woven class:

```
public class Invoice {
  public virtual decimal GrandTotal() {
    decimal total = ... complex computation ... ;
    return total * (total < 10000 ? 1.0M : 0.95M);
  }
}
```

Further weaving of class Invoice

- A charity should get everything for free (and such gifts are deductible)

```
public class MyNewInvoiceAspect {
  private bool charity;
  private decimal deductible;

  public bool Charity { set { charity = value; } }

  public decimal CharityAspect() {
    decimal total = JoinPointContext.Proceed<decimal>();
    if (charity) {
      deductible += total;
      total = 0;
    }
    return total;
  }
}
```

Pointcut file for further weaving

- Inserts fields **`charity`** and **`deductible`**
- Inserts set-property **`Charity`**
- Intercepts method **`GrandTotal`**

```
insert field private instance bool
  MyNewInvoiceAspect:charity into Invoice;
insert field private instance decimal
  MyNewInvoiceAspect:deductible into Invoice;
insert method public instance void
  MyNewInvoiceAspect:set_Charity(bool) into Invoice;
around * * decimal Invoice:GrandTotal()
  do MyNewInvoiceAspect:CharityAspect;
```

Customization as aspects?

versions

4.0

3.0

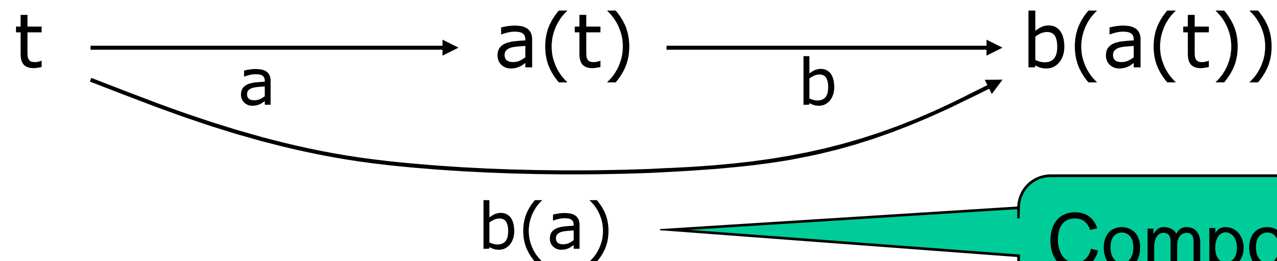
Kernel

Customization by aspect weaving

...mizations

Advice composition (new?)

- Let a(t) be advice a applied to target t
- The result is code and can be further woven as in b(a(t))
- Advice is normal code; can be woven
- So advice composes b(a(t))=(b(a))(t)



- If feature=aspect, then features compose
- But do not commute: b(a(t)) ≠ a(b(t))

Evaluating the aspect approach

- Observations:
  - Aspects compose, but order matters
- Advantages of aspects
  - Static typechecking of customizations:
    - before weaving, when compiling the advice;
    - at weaving time; and
    - bytecode pre-verification (peverify) before deployment
  - No runtime or space overhead
  - Can remove, add and change state and behaviour
- Disadvantages:
  - Works on implementation, not specification: class names and method signatures matter (too much)
  - The "interface specification" could be conventions for naming and argument types – but this is brittle
  - Still requires foresight on possible customizations

Code duplication

- As software evolves, architecture decays
- Code duplication (clones) is one sign of this
- New functionality similar to old functionality is made by copy/paste/edit
- Because maintainers
  - don’t fully understand the old code (didn’t write it)
  - are afraid of breaking it if generalizing (refactoring)
  - don’t want to invest much time in the new code until they know what it *should* do (no clear specification)
  - and so they do “prototyping” by duplication, but forget to clean up (next deadline looms)
- Clone detection is a way to find new variation points, for refactoring
- Also useful in the MS Dynamics partner model

Code duplication discovery

- Baker (1993): parametrized clone detection
- Schou (2007): Baker's ideas implemented for C#, with clone grouping and visualization
- Code normalization; abstraction of names and literals; suffix trees; clone clustering; dotplots
- Empirical study, preliminary results:
  - Danish insurance software company, extremely profitable, 80 well-educated developers
  - Core project 1.4 mill lines C# in 4346 files, 6 years
  - Code duplication analysis takes 80 secs on a laptop
  - Many clone groups, including a 258 line switch
  - Many clones are user interface related:
    - Create a dialog with slightly changed text labels etc.
  - Maybe make GUI more data-driven: commonalities in code, configuration by data?

Future work?

- Aspect weaving for .NET
  - Some unique ideas in the Yiihaw weaver
  - Still lacks support for some C# features
  - Type rules too strict for the expression problem
  - Future PhD project?
- Aspects for customization?
  - Is this approach powerful enough for AX?
  - Advantages over text-based customization?
- Code duplication discovery
  - Schou's tool seems very useful
  - Could be adapted to X++ or C/AL rather easily
  - Better visualization of file-file comparisons desirable