Frameworks / Product Lines

The frame for reuse

Motivation

I have presented some design principles

- 3-1-2 process: *encapsulate variation, program to an interface and compose behaviour using delegation*
- Compositional design
  - it is better to reuse code by delegation than inheritance
  - *let someone else do the dirty job*
- Iterative and Incremental search for variability
  - supported by TDD
  - supported by refactoring

... and on the way we discovered some patterns

Abstracting

What I have actually done is to show you the pieces that together form the puzzle called *frameworks:*

Framework:

- A framework is a set of cooperating classes that make up a reusable design for a specific class of software. [GoF]

Exercise:

- Is the pay station a framework given this definition?
- Argue why MiniDraw is a framework.

Framework Characteristics

A framework is characterized by

- reuse of *both* design as well as executable code
- reuse within a well defined domain
  - Swing is for GUI's, not for banking…
- high reuse percentage (70-90%)
- must be customized for customer's particular need

Example:

- HP printer software
- Eclipse IDE plug-in architecture
- CelciusTech C&C for ships
- Java Swing

Other Definitions

- A framework is: a) a reusable design of an application or subsystem b) represented by a set of abstract classes and the way objects in these classes collaborate (Fraser et al. 1997).
- A framework is a set of classes that embodies an abstract design for solutions to a family of related problems (Johnson and Foote 1988).
- A framework is a set of cooperating classes that make up a reusable design for a specific class of software (Gamma et al. 1995, p. 26).
- A framework is the skeleton of an application that can be customized by an application developer (Fayad et al. 1999a, p. 3).
- A framework defines a high-level language with which applications within a domain are created through specialization (Pree 1999, p. 379).
- A framework is an architectural pattern that provides an extensible template for applications within a domain (Booch et al. 1999, p. 383).

Common aspects of the definitions

Skeleton / design / high-level language

- … provide behaviour on **high level of abstraction:** a design/skeleton/architecture

Application/class of software

- … has a **well defined domain** where it provides behaviour

Cooperating / collaborating classes

- … define and limit **interaction patterns (protocol) between well defined roles**

Customize/abstract classes/reusable/specialize

- … can be **tailored** to a concrete context

Classes/implementation/skeleton

- **...** is reuse of **code** as well as reuse of design

Variability points = Hotspots

Applications made from frameworks

A framework based application is composed of *three* portions of code

- The framework code (external, third party)
- The framework customization code (our own code)
- Non-framework related code (own code)

Example: Backgammon project

- MiniDraw framework (basic GUI control)
- Customization code (adapt MiniDraw til BG graphics)
- Backgammon domain code (move validation, turn control, dice rolling, etc.)

Which leads to...

Traditionally one distinguish between

- Developers: Developing the software for end users
- End users: Using the software

Frameworks require one additional role

- Framework developers: Develops the framework
- Application developers: Adapt the FW for users
- End users: Using the software

I.e.: *The end users of a framework are themselves software developers!*

HotSpots…

”Separate code that changes from the code that doesn’t”

Definition: Frozen spot

A part of framework code that cannot be altered and defines the basic design and the object protocols in the final application.

Definition: Hot spot

A clearly defined part of the framework in which specialization code can alter or add behavior to the final application.

Hot spots are also known as:

- **Hooks / hook methods**
- **Variability points (our favourite term)**

Example

The pay station system has **hotspots** regarding

- receipt type
- rate policy
- display output
- weekend determination

but **frozen spots**, fixed behaviour, concerning

- coin types (payment options), numerical only display, only buy and cancel buttons, etc.

Domain: Pay stations

- no reuse in the electronic warfare domain ☺

Frozen spots

Frozen is important : keep the code in the fridge!

Key Point: Frameworks are not customized by code modification

A framework is a closed, blackbox, software component in which the source code must not be altered even if it is accessible. Customization must only take place through providing behavior in the hot spots by those mechanisms laid out by the framework developers.

Code modifications means unthawing and freezing again ☠ ☠ ☠

Why...

I have realized that this point is so natural to me that I have more or less forgotten to emphasize it.

Why

- Consider Java Swing
  - If you had to *rewrite code in 8 different places in the Swing library to add a new special purpose visual component*
- Then you had to
  - Understand the Swing code in deep detail (costs!)
  - Reintroduce special purpose code everytime Sun/Oracle releases a new version of Swign (costs!!!)

Morale

You adapt a framework to your particular needs by

- *Change by addition, not by modification!!!*

You

- Implement interfaces or extend existing classes
  - Concrete implementations, **real behaviour filling out the responsibilities defined by the framework**
- Tell the framework to use *your* implementations...

Which again leads to...

If the framework has to use my concrete implementations then...

It cannot itself create these objects

- If MiniDraw itself had
  - Drawing = new StandardDrawing();
- Then it was an application (fixed behaviour), not a framework (adaptable behaviour).

Then ... How can it create objects?

Dependency Injection

Key Point: Frameworks must use dependency injection

Framework objects cannot themselves instantiate objects that define variability points, these have to be instantiated by the application code and injected into the framework.

Typically an (OO) framework would use factory techniques like **abstract factory**

- Or prototype or factory method patterns...

AARHUS UNIVERSITET

Customization Techniques

Henrik Bærbak Christensen

Techniques to define hotspots

We have object oriented composition to define hotspots. We have also looked at other techniques

- Parameterization, overriding superclass methods

HotSpots

Exercise: List techniques to *make the hotspots*

- Hint: Consider techniques used in
  - MiniDraw
  - Swing
  - Java RMI
  - Collection classes
  - Eclipse

  - C library sorting algorithms
  - Intel 486 interrupt vector handling
  - Functional languages

Does a Framework require object-orientation?

A good framework...

... Must give ”return on investment”

- Investment: ”I have to learn how to use it”
- Return: ”I get reliable software that does a lot”

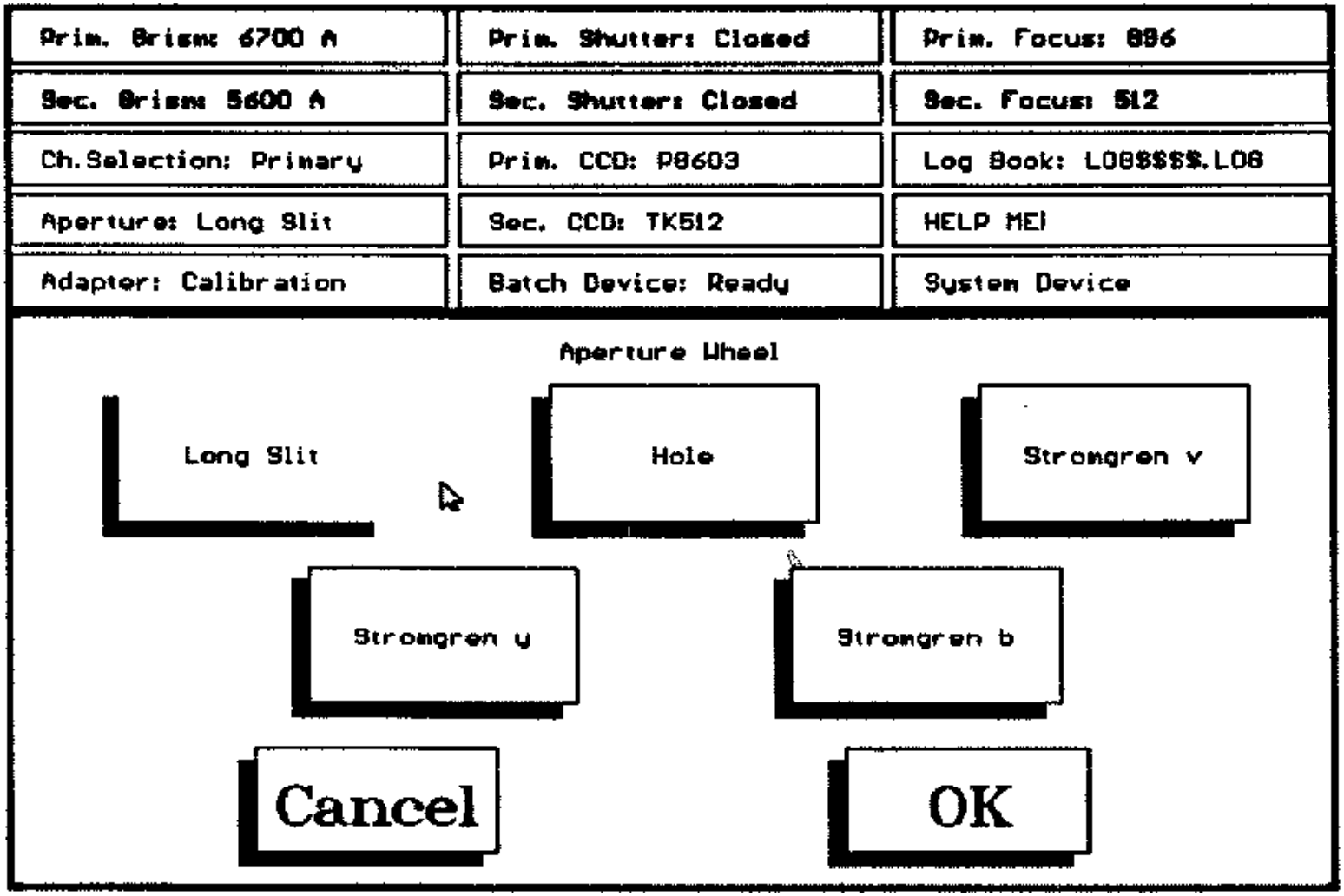
Swing/MiniDraw/...

- Consider writing Swing yourself!

Key Point: Frameworks should support the spectrum from no implementation (interface) over partial (abstract) to full (concrete) implementation for variability points

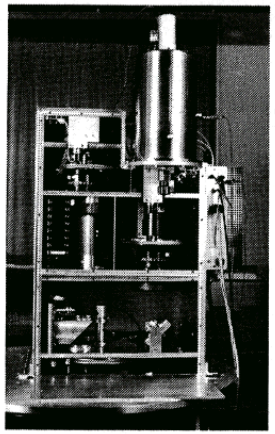
A framework provides the optimal range of possibilities for the application developer if all variability points are declared in interfaces, if these interfaces are partially implemented by abstract classes providing “common case” behavior, and if a set of concrete classes for common usages is provided.

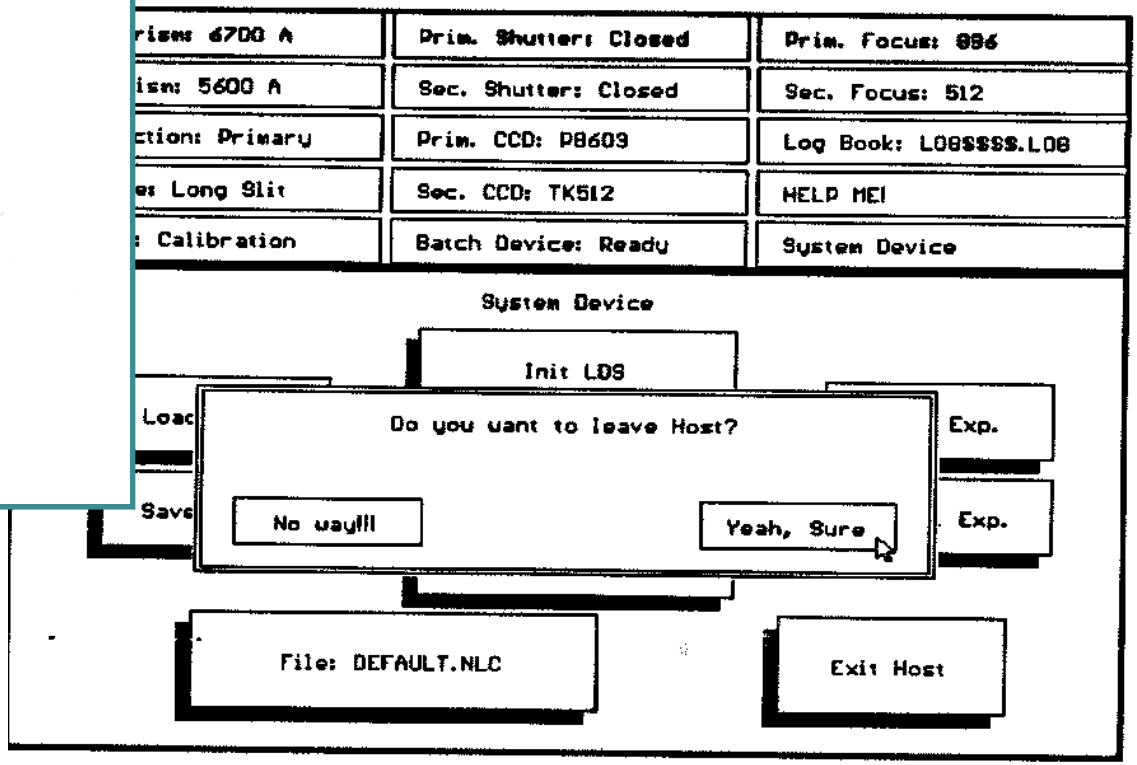
In the old times...



A Control System for the
Aarhus-Tromsø
Low Dispersion Spectrograph.

Henrik Bærbak Christensen
Institute of Astronomy
University of Aarhus





Framework Protocol

Design *and* Code Reuse

"Cooperating / collaborating classes

- … define and limit **interaction patterns (protocol) between well defined roles"**

Frameworks require users (=developers) to understand the interaction patterns between objects in the FW and their own customization objects. That is, understand the **protocol.**

- Ex: Understand the editor↔tool protocol in MiniDraw
- One TA tried to invoke *activate* on a tool (and ended with an infinite loop)
  - *activate is a method called by MiniDraw when a tool is activated!*

Where is the protocol?

So: The framework **dictates** the protocol!

The question is: How?

Protocol

The protocol arise because *objects in the framework invokes methods on objects that are defined by you.*

These methods/functions/procedures that are predefined by the framework to be ‘overridable’ or customizable are the *hotspots.*

A framework contains *frozen code* with embedded *hotspots* where your (unfrozen) code may be called.

Inversion of Control

This property of frameworks is called

Inversion of Control

(”Hollywood principle: *do not call us, we will call you.)*

The framework dictates the prototocol – the customization code just has to obey it!

Compare ‘traditional’ reuse

Another type of reuse of code (more than design) is *libraries*

I have never written my own *cosine* function – have you?

There are lot of libraries of usable behavior out there that does not *invert control.*

Inversion of Control

Note: inversion of control is not a 'required' characteristics of a framework in some author's view

- RMI is an example. The control inversion is difficult to spot...
- (but why is it not just a library then???)

Template Method

The central OO pattern to separate frozen and hot code

The core of the *inverted control*

Template Method

Intent (Original GoF statement)

- *Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.*

Restatement

- Some steps in the algorithm are fixed but some steps I would like to keep open for future modifications to the behaviour

GoF Structure

The structure in GoF is that of *subclassing*

```
abstract class AbstractClass {
  public void templateMethod() {
    [fixed code part 1]
    step1();
    [fixed code part 2]
    step2();
    [fixed code part 3]
  }
  protected abstract void step1();
  protected abstract void step2();
}
class ConcreteClass extends AbstractClass() {
  protected void step1() {
    [step 1 specific behavior]
  }
  protected void step2() {
    [step 2 specific behavior]
  }
}
```

The Multi-Dimensional Variability

Subclassing handles multi-dimensional variability very badly...

Consider

- Three variants of step1() required
- Four variants of step2()
- Any combination feasible

How many subclasses?

```
abstract class AbstractClass {
  public void templateMethod() {
    [fixed code part 1]
    step1();
    [fixed code part 2]
    step2();
    [fixed code part 3]
  }
  protected abstract void step1();
  protected abstract void step2();
}
class ConcreteClass extends AbstractClass() {
  protected void step1() {
    [step 1 specific behavior]
  }
  protected void step2() {
    [step 2 specific behavior]
  }
}
```

Conclusion

Conclusion:

Favour object composition over class inheritance

Exercise

Rewrite the GoF structure to its compositional equivalent that is behavioural equivalent but follows the three *principles of flexible design.*

```
abstract class AbstractClass {
  public void templateMethod() {
    [fixed code part 1]
    step1();
    [fixed code part 2]
    step2();
    [fixed code part 3]
  }
  protected abstract void step1();
  protected abstract void step2();
}
class ConcreteClass extends AbstractClass() {
  protected void step1() {
    [step 1 specific behavior]
  }
  protected void step2() {
    [step 2 specific behavior]
  }
}
```

Find the error in the Book

The book's code is not wrong but it misses the point in supporting multi-dimensional variability ☹

Improved version:

```
class Class {
  private HookInterface1 hook1;
  private HookInterface2 hook2;
  public void setHook( HookInterface1 hook1,
                       HookInterface2 hook2) {
    this.hook1 = hook1;
    this.hook2 = hook2;
  }
  public void templateMethod() {
    [fixed code part 1]
    hook1.step1();
    [fixed code part 2]
    hook2.step2();
    [fixed code part 3]
  }
}
interface HookInterface1 {
  public void step1();
}
interface HookInterface2 {
  public void step2();
}
class ConcreteHook1 implements HookInterface1() {
  public void step1() {
    [step 1 specific behavior]
  }
class ConcreteHook2 implements HookInterface2() {
  public void step2() {
    [step 2 specific behavior]
  }
}
```

Template Method Terminology

These two variants have a name:

- Original: subclassing+override hook methods
- New: interface implementation + delegate to hooks

Definition: Unification

Both template and hook methods reside in the same class. The template method is concrete and invokes abstract hook methods that can be overridden in subclasses.

Definition: Separation

The template method is defined in one class and the hook methods are defined by one or several interfaces. The template method is concrete and delegates to implementations of the hook interface(s).

Exercise

Identify template method in the pay station:

```
public void addPayment( int coinValue ) {
   switch ( coinValue ) {
   case 5:
   case 10:
   case 25: break;
   default:
     throw new IllegalCoinException("Invalid coin: "+coinValue+" cent.");
   }
   insertedSoFar += coinValue;
   _timeBought = rateStrategy.calculateTime(insertedSoFar);
 }
```

Exercise

How does *template method* relate to the *inversion of control* property of frameworks?

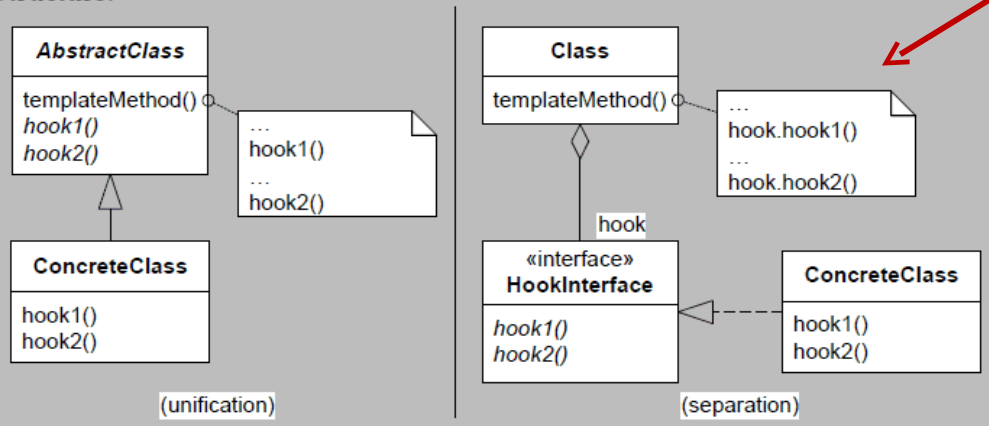
[31.1] Design Pattern: Template Method

Intent Define the skeleton of an algorithm in an operation, deferring some steps to subclasses or delegates. Template Method lets the behavior of certain steps of an algorithm be varied without changing the algorithm's structure.

Problem There is a need to have different behaviors of some steps of an algorithm but the algorithm's structure is otherwise fixed.

Solution Define the algorithm's structure and invariant behavior in a template method and let it call hook methods that encapsulate the steps with variable behavior. Hook methods may either be abstract methods in the same class as the template method, or they may be called on delegate object(s) implementing one or several interfaces defining the hook methods. The former variant is the *unification* variant, the latter the *separation* variant.

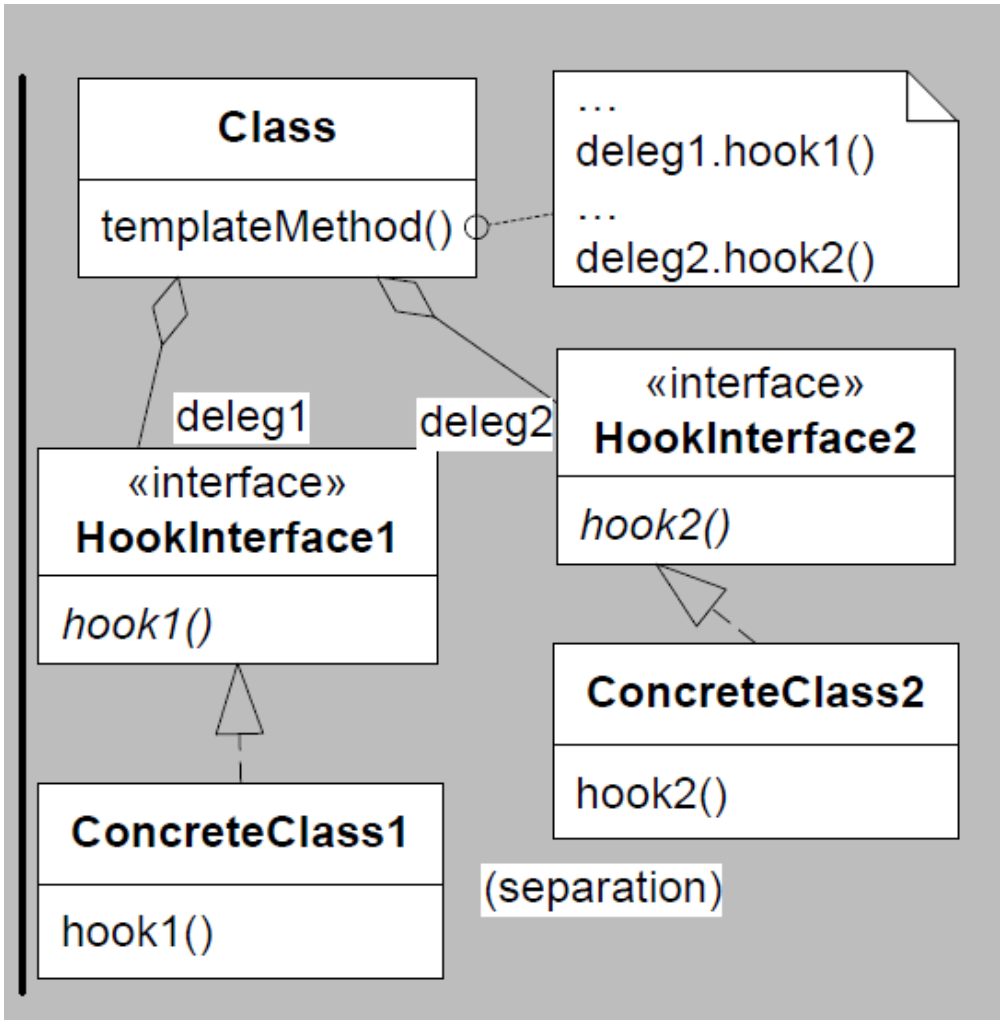
Structure:



Roles The roles are method abstractions: the **template method** defines the algorithm structure and invariant behavior. **Hook methods** encapsulate variable behavior. The **HookInterface** interface defines the method signatures of the hook methods.

Cost - Benefit The benefits are that the *algorithm template is reused* and thus avoids multiple maintenance; that the *behavior of individual steps, the hooks, may be changed*; and that *groups of hook methods may be varied independently* (separation variant only). The liability is *added complexity of the algorithm* as steps have to be encapsulated.

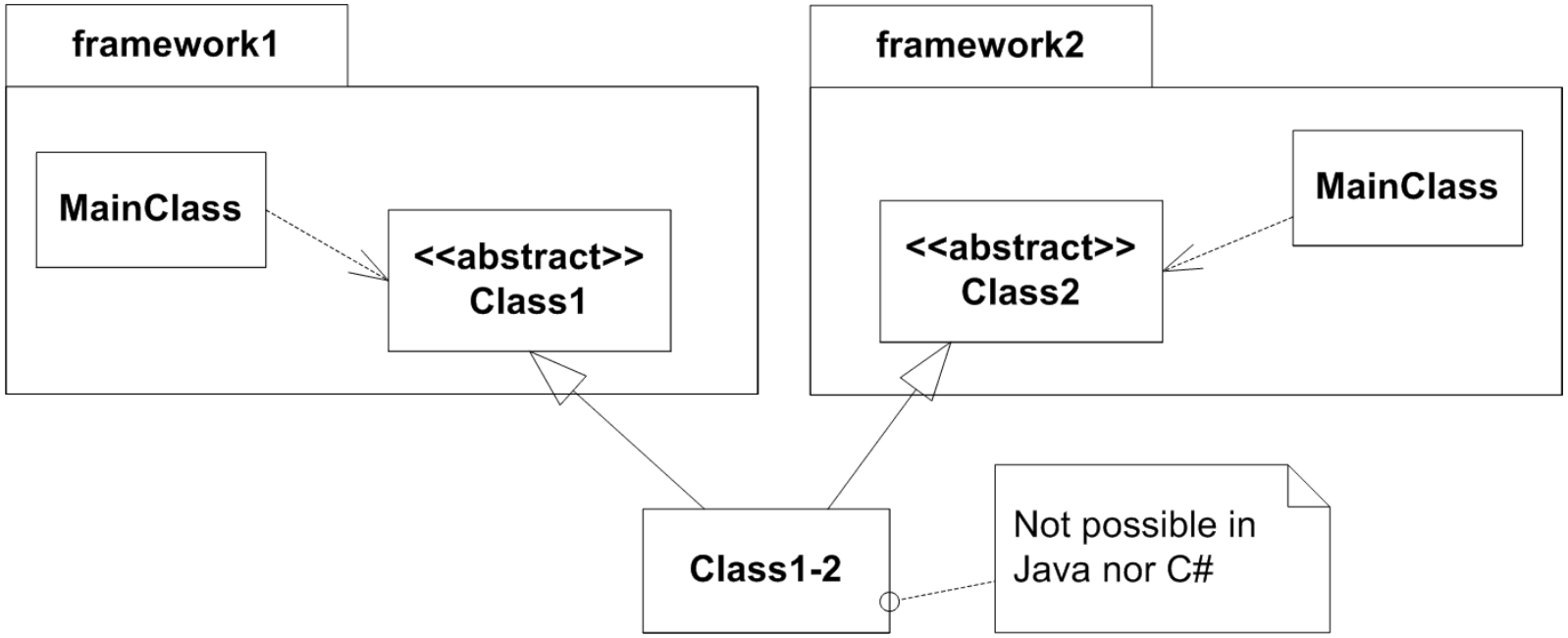
Improved structure



Another Case for Composition

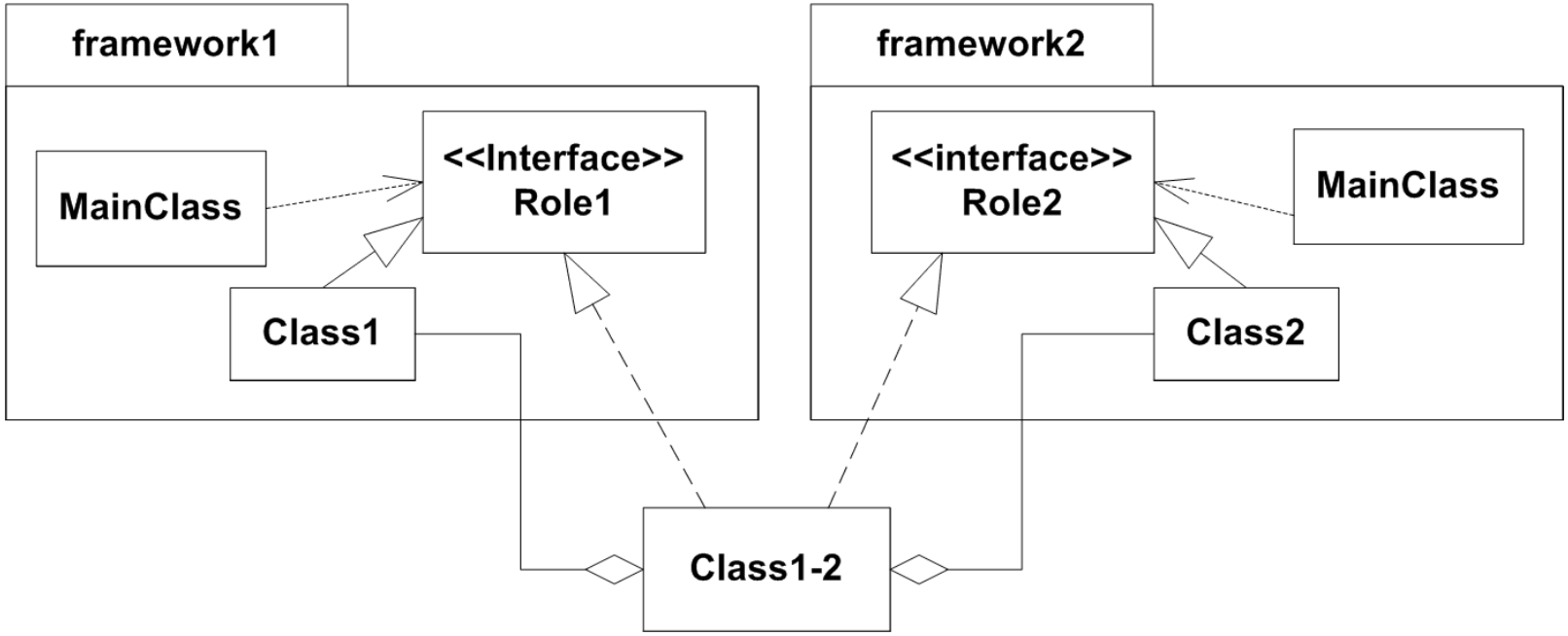
Mixing Two Frameworks

Based upon the inheritance based template method



Mixing Two Frameworks

But it does work using compositional template method



Software Reuse and Product Line Architectures

Reuse of software

Software reuse

- High quality (= reliable) software
  - If mature and well supported
- Lower cost
  - One line of reliable implementation is expensive!!!
  - Easier to buy than to develop
- Faster development
  - Building apps is faster and cheaper

Sidebar 32.1: MiniDraw Reuse Numbers

MiniDraw is approximately 1,600 lines of code (LOC) including comments, and the jar file about 30KB. The numbers for the three demonstration applications are: puzzle 55 LOC, rect 200 LOC, and marker 125 LOC. Thus if I calculate the percentage of lines of code that is reused in the final applications I get:

Puzzle: 97 % Rect: 89 % Marker: 93 %

Reuse types

There are many different types of reuse:

- Math.cos(double x) **code reuse**
- Strategy Pattern **design reuse**
- Knuth's binary search alg. **design reuse**

Frameworks are pretty special:

Key Point: Framework reuse is reuse of both design and code

A framework is a tangible unit of software, thus it is code reuse. However, due to the inversion of control property, it also defines the flow of control, object protocols, and clearly defines the variability points, and thus bundles a quality design as well.

Software Engineering Institute

SEI in Pittsburgh

– (associated with Carnegie Mellon University)

Definition: Software product line

A *software product line* is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. (Bass et al. 2003, p. 353)

Analysis

Definition: Software product line

A *software product line* is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. (Bass et al. 2003, p. 353)

Characteristics

- software intensive system
- common, managed, set of features
- needs of particular market segment
- developed from a common set of core assets
- prescribed way

Exercise: Compare PayStation

Framework/Product Line

In my view the *technical aspects* of a product line is covered by the framework concept

- A framework is a set of cooperating classes that make up a reusable design for a specific class of software.

Definition: Software product line

A *software product line* is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. (Bass et al. 2003, p. 353)

Organisation Aspect

The *organisation aspect*, however, is important:

- *managed* set of features
- developed in a *prescribed way*
- *market segment*

A lot of experience from industry point to the fact that software reuse and product line reuse is *primarily a organisational challenge!*

- *Who pays for making the code flexible? Not my project – I want to save my own a…*

Reuse is actually quite expensive in practice ☹