A. WHAT'S A FRAMEWORK?
A SOFTWARE FRAMEWORK

A particular implementation technique for building families of software applications.

A framework represents a common design and partial implementation for the family:

A *generic solution* for a set of similar problems.

*Incomplete by nature* : application-specific functionality to be filled in by the *framework customiser*, i.e. the developer of a concrete application.

*Variations* are specified by means of so-called *hot spots*. 
FRAMEWORKS ARE ABOUT SOFTWARE REUSE

Frameworks are meant to be reused.

Designing a framework is not easy:

A good framework should be easy to use and be flexibly adapted to a wide range of requirements.

Identifying the right combination of hot spots is difficult.

Best achieved via an iterative development process.

Need at least 3 applications before turning it into a framework.
An object-oriented (application) framework is an object-oriented class hierarchy plus a built-in model of interaction which defines how objects derived from the class hierarchy interact with one another.

Deriving a custom application from a framework is typically done through class specialisation.

OBJECT-ORIENTED APPLICATION FRAMEWORKS

Support reuse beyond the class level.

Core functionality implemented as set of abstract classes that cooperate in a well-defined manner.

When deriving a concrete application:

- these abstract classes are specialised by concrete subclasses;
- other concrete classes are chosen from a library of standard components provided by the framework developer.

Customisation is completed by adding new application-specific classes.
EXAMPLES OF FRAMEWORKS

GUI frameworks (e.g., JHotDraw)
Unit testing frameworks (e.g., JUnit)
Collection hierarchy (e.g., Smalltalk or Java)
A particular MVC implementation
Web application frameworks

WHATS'On, an application framework for television broadcast management

YESPLAN, an application framework for event planning
SOME DEFINITIONS OF FRAMEWORKS

[Ralph Johnson, OOPSLA 97] : “A reusable design of an application or subsystem, represented by a set of abstract classes and the way objects in these classes collaborate.”

[GoF p. 26] : “A set of co-operating classes that make up a reusable design for a specific class of software.”

[Fayad et al. §1] : “The skeleton of an application that can be customised by an application developer.”

[Fayad et al. §16] : “Defines a high-level language with which applications within a domain are created through specialisation.”

[Van Gurp & Bosch] : “A partial design and implementation for an application in a given domain.”
CENTRAL ASPECTS IN THESE DEFINITIONS

Domain / class of software: has a well defined domain where it provides behaviour.

Skeleton / design / high-level language: a common design shared by all customisations.

Collaborate / co-operating: a description of the behaviour at a high level of abstraction, defining how classes participating in the framework interact.

Reusable / abstract classes / customised / specialisation: can be tailored to a concrete context.

Classes / partial implementation: reuse of code as well as reuse of design
FRAMEWORK TYPES: APPLICABILITY

Domain frameworks capture expertise useful for one particular problem domain:

- financial engineering
- television broadcast management
- event planning

Application frameworks capture expertise common to a wide variety of problems:

- graphical user interface frameworks
- collection classes
- web application frameworks
WHY FRAMEWORKS?

Frameworks are one of the best bets on Software Reuse

High-level design is the main intellectual content of software, and frameworks are a way to reuse it...

Frameworks allow you to reuse both design and implementation

“Interface design and functional factoring constitutes the key intellectual content of software and are far more difficult to create or re-create than code.”

[Peter Deutsch]
THE DIFFERENT PARTS OF A FRAMEWORK-BASED APPLICATION

An application consists of:

- **The framework code itself** provided
  
e.g. JHotDraw

- **The framework specialisation code** your job!
  
e.g. JHotDraw specialisation to handle musical notation

- **... and the rest** your job!
  
drivers, utilities, application parts not handled by the framework....

- **e.g. code to handle musical semantics, playback, etc.**
FRAMEWORK DEVELOPMENT = "PROGRAMMING WITH HOLES"

A framework is a partial application

The framework

Hole

Hole

Hole

Hole
FRAMEWORK DEVELOPMENT = "PROGRAMMING WITH HOLES"

A framework is a partial application

Your application

Ken's code

Your code

Jim's code

Some more code
PRINCIPLE OF INVERSION OF CONTROL

a.k.a. the Hollywood principle:

“Don’t call us, we’ll call you”

This is what distinguishes a framework from a library

When using a library, the application calls the library, but the library is not aware of the application.

When using a framework, the application-specific code written by the programmer gets called by the framework.
PRINCIPLE OF INVERSION OF CONTROL

Frameworks are partial applications and thus (usually) define interaction patterns. Thus they insist on defining the flow of control:

Coordination happens in application code

Coordination happens in the framework code
HOTSPOTS

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

Hotspots are the “holes” of a framework

- Code points where specialisation code can alter behaviour or add behaviour to the framework

Also known as: hooks / hook methods / variation points

Commonality / variability

- The framework code defines the commonality
- The hotspots allow for variability
Filling in hotspots by specialising abstract classes, methods and interfaces
HOTSPOTS BASED UPON COMPOSITION

```
a := SavingsAccount new.
a associate: MasterCard new.
a associate: VISA new.
...
```

_Filling in parameters or objects by prefabricated components_
FRAMEWORK TYPES: CUSTOMISATION

White-box frameworks

Customisation through inheritance

Require insight in (and access to) implementation

“Easier” to design

More difficult to learn

More programming required for application development

More flexibility
FRAMEWORK TYPES: CUSTOMISATION

Black-box frameworks

Customisation through composition

Require insight in provided components

“Harder” to design

“Easier” to learn

Less programming required for application development

Limited flexibility (no unanticipated variations)
FRAMEWORK TYPES: CUSTOMISATION

Grey box frameworks

White and black box form the extreme boundaries of framework design and usage principles.

Most frameworks live somewhere in between these two extremes.

Grey box frameworks attempt to realise the benefits of both white and black box designs, while trying to avoid the perceived limitations of both.

A successful framework may start its life as white box, maturing towards grey or even black in a number of revisions.
CONCRETE EXAMPLE

... OF A FRAMEWORK

**Figure {abstract}**
- currentpos
- move(newpos)
- draw(pos) {abstract}

**Point**
- x, y
- draw(pos)

**Line**
- Point p1, p2
- draw(pos)

**Polygon**
- Collection<Point>
- draw(pos)

**Circle**
- center, radius
- draw(pos)

**GroupedFigure**
- Collection<Figure> figs
- draw(pos) {
  for each f in figs :
  f.draw(pos) 
}
CONCRETE EXAMPLE

... OF A FRAMEWORK

... OF AN APPLICATION
CONCRETE EXAMPLE

... OF A FRAMEWORK

... OF AN APPLICATION

**Figure** {abstract}
- `currentpos`
- `move(newpos)`
- `draw(pos)` {abstract}
...

- **GroupedFigure**
  - `Collection<Figure> figs`
  - `draw(pos) {abstract}
    for each f in figs : f.draw(pos) }`

- **Point**
  - `x, y`
  - `draw(pos)`

- **Line**
  - `Point p1, p2`
  - `draw(pos)`

- **Polygon**
  - `Collection<Point>`
  - `draw(pos)`

- **Circle**
  - `center, radius`
  - `draw(pos)`

- **Clef**
  - `draw(pos)`

- **LedgerLines**
  - `draw(pos)`

- **Note**
  - `draw(pos)`
B. TEMPLATE METHODS (INTERLUDIUM)
TEMPLATE METHOD DESIGN PATTERN

Intent

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

Solution

Break out primitive steps into separate methods in ancestor class.

Construct method for basic algorithm in ancestor that calls the primitive methods.

Override the primitive methods in descendant classes to implement specific tasks.
Consequences

A fundamental technique for code reuse - particularly important in class libraries and frameworks to factor out common behaviour.

Leads to inverted control structure called “Hollywood Principle”.

A primitive method in the ancestor may provide a default behaviour that descendants may optionally override (called hook methods).

Related Patterns

*Factory Method* is a form of Template Method used to create families of related objects.

*Strategy* is an alternate choice when the behaviour needs to be specified or may vary at run-time.
A **hotspot** in an object-oriented framework is often implemented via a Template Method.

The **template method** defines the skeleton of the hot spot.

The variable parts are deferred to the so-called **hook methods**.

The template method is defined on a template class which is part of the framework.

The hook methods are defined on **hook classes**. These are concrete subclasses of the template class that are provided by framework users to customise the framework.
**TEMPLATE METHOD IN FRAMEWORKS**

**Abstract class**

- `templateMethod(arg)`
- `hookMethod1`
- `hookMethod2`

The framework

Framework user = a customiser

(a developer of a concrete application)
The application customises a framework’s hot spots.
The framework provides abstract classes that must be customised before they can be used.

The application provides subclasses that customise the template methods by implementing the (abstract) hook methods.
**Template Method Design Pattern**

**Abstract class**
- `templateMethod(arg)`
- `hookMethod1`
- `hookMethod2`

**Specific class**
- `hookMethod1(arg)`
  - "do something concrete with arg"
- `hookMethod2`
  - "do something concrete"

```java
abstract class TemplateMethod {
    templateMethod(arg) {
        ...
        this.hookMethod1(arg)
        ...
        this.hookMethod2
        ...
    }
}
```
**Concrete Example**

**Figure {abstract}**
- currentpos
- move(newpos)
  - undraw() {abstract}
  - draw(pos) {abstract}

**Triangle**
- Point p1, p2, p3
- draw(pos)
- undraw(pos)

```java
move(newpos) {
    this.undraw();
    this.draw(newpos);
}
```

```java
draw(pos) {
    "draw this Triangle at position pos"
}
```

```java
undraw(pos) {
    "remove this figure at its position currentpos"
    ...
}
```
SUMMARY

Template methods are a key technique for building OO application frameworks.

Methods as units of reuse

Inheritance as parametrisation mechanism

Late binding of self

self is dynamic; acts as a hook
C. FRAMEWORKS (CONTINUED)
CONTRACT BETWEEN FRAMEWORK AND APPLICATION DEVELOPER

Framework (developer) must:

- Provide expensive domain knowledge and design
- Provide concrete, reliable, executable software
- Be sufficiently flexibility to specialise for required context.
- Be usable and “easy” to learn (this is a non-trivial requirement)

Application (developer) must:

- keep the contracts of hotspots
- understand and follow the interaction rules
LEARNING FRAMEWORKS

Understanding a framework is vital for success

- more difficult to understand abstract entities than concrete classes
- interaction patterns are ‘hidden’ but vital in order to use a framework correctly

Learning a framework is not easy

- Steep learning curve
- Black box frameworks easier to learn than white box
FRAMEWORK COMPOSITION...

How to use more than one framework in a single application?

Diagram:

- Banking Framework
  - ATM
  - BankCard
  - <<abstract>> Account
  - uses

- Insurance Framework
  - Contract
  - Beneficiary
  - <<abstract>> Insurance
  - uses

- TheSuperClass
  - Insurance-Account
FRAMEWORK COMPOSITION...

A possible solution: using delegation.

The diagram illustrates the composition of object-oriented application frameworks, showing the relationships between the Banking Framework, Insurance Framework, and their components such as Account, BankCard, Contract, Insurance, Beneficiary, DefaultAccount, and Default-Insurance. The diagram also highlights the use and delegation relationships among these components.
OBJECT-ORIENTED APPLICATION FRAMEWORKS

SUMMARY

- Frameworks
  - partial / skeleton application within a well-defined domain
  - can be tailored / customised for a specific application
  - reuse of implementation and design
- Inversion of Control ("Hollywood" principle)
  - framework defines flow and interaction patterns
- Hotspots = ‘hooks’ into framework where tailoring is made
  - inheritance based : white box approach
  - composition based : black box approach
- Commonality and variability
- Use of template methods as implementation technique
FURTHER READING

Object-Oriented Application Frameworks
Ted Lewis and friends
Manning Publications, 1995

Building Application Frameworks: Object-Oriented Foundations of Framework Design
Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson
John Wiley & Sons, 1999

Java Application Frameworks
Darren Govoni, John Wiley & Sons, 1999
D. DESIGN PATTERNS VS. FRAMEWORKS
DESIGN PATTERNS VS FRAMEWORKS

Both frameworks and design patterns are ways of describing and documenting solutions to common problems.

But design patterns are not frameworks.

Patterns are more abstract.

And many patterns may be involved in the solution of one problem.
DESIGN PATTERNS VS FRAMEWORKS

Frameworks

codify designs for solving a family of problems within a specific domain

are instantiated by inheritance and composition of classes

can contain several instances of multiple design patterns

are more “shrink-wrapped”, ready for immediate use
Design patterns are more abstract, smaller architectural elements, less specialised than frameworks.
E. Refactoring to a Framework
THREE CATEGORIES OF REFACTORINGS RELATED TO FRAMEWORK DEVELOPMENT

Three categories of refactorings

that correspond to generic design evolutions occurring frequently in object-oriented software systems. [Demeyer&al 2000]

1. Create template methods
2. Optimise class hierarchies
3. Incorporate composition relationships

THREE CATEGORIES OF REFACTORINGS RELATED TO FRAMEWORK DEVELOPMENT

1. Create template methods

Split methods into smaller chunks to separate common behaviour from specialised parts so that subclasses can override.

Used to improve reusability, remove duplicated functionality.
THREE CATEGORIES OF REFACTORINGS RELATED TO FRAMEWORK DEVELOPMENT

2. Optimise class hierarchies

Insert or remove classes within a class hierarchy and redistribute the functionality accordingly.

Used to increase cohesion, simplify interfaces, remove duplicated functionality.

Two subcategories:

A. refactor to *specialise*

B. refactor to *generalise*
2. OPTIMISING CLASS HIERARCHIES

A. Refactor to specialise

Improve framework design by decomposing a large, complex class into several smaller classes.

The complex class usually embodies both a general abstraction and several different concrete cases that are candidates for specialisation.
**2A. REFACTOR TO SPECIALISE : EXAMPLE**

Disc Management for NTFS

<table>
<thead>
<tr>
<th>Disc</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>copyDisc</td>
</tr>
<tr>
<td>formatDisc</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

Disc Management for NTFS & OSX

<table>
<thead>
<tr>
<th>Disc</th>
</tr>
</thead>
<tbody>
<tr>
<td>disctype</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>copyDisc</td>
</tr>
<tr>
<td>formatDisc</td>
</tr>
</tbody>
</table>

formatDisc

self discType = #NTFS
ifTrue: [ .. code1 ..].
self discType = #OSX
ifTrue: [ .. code2 ..].

Disc Management for NTFS & OSX

<table>
<thead>
<tr>
<th>Disc</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>copyDisc</td>
</tr>
<tr>
<td>formatDisc</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

NTFSDisc

| copyDisc |
| formatDisc |
| ... |

OSXDisc

| copyDisc |
| formatDisc |
| ... |
Specialise a class by adding subclasses corresponding to the conditions in a conditional expression:

Choose a conditional whose conditions suggest subclasses (this depends on the desired abstraction).

For each condition, create a subclass with a class invariant that matches the condition.

Copy the body of the condition to each subclass, and in each class simplify the conditional based on the invariant that is true for the subclass.

Specialise some (or all) expressions that create instances of the superclass.
B. Refactor to generalise

Identify proper abstractions (e.g. abstract classes) by examining concrete examples and generalising their commonalities.

Diagram:

- Concrete class A
- Concrete class B
- Abstract class X

Commonalities:

- concrete class A
- concrete class B
2B. REFACTOR TO GENERALISE

Abstract classes and frameworks are generalisations.

People think concretely, not abstractly.

Abstractions are found bottom up, by examining concrete examples first.

Generalisation proceeds by:

- finding things that are given different names but are really the same (and thus renaming them)
- parameterisation to eliminate differences
- breaking large things into small things so that similar components can be found
2B. REFACTOR TO GENERALISE

Steps to create an abstract superclass:

- Create a common superclass
- Make method signatures compatible
- Add method signatures to the superclass
- Make method bodies compatible
- Make instance variables compatible
- Move instance variables to the superclass
- Move common code to the abstract superclass
2B. REFACTOR TO GENERALISE: EXAMPLE

abstract commonalities

PrintServer
print

FileServer
save

OutputServer
output

PrintServer

FileServer
THREE CATEGORIES OF REFACTORINGS RELATED TO FRAMEWORK DEVELOPMENT

3. Incorporate composition relationships

Move functionality to (newly created) sibling classes.

Used to reduce coupling, migrate towards black-box frameworks.

Motivation:

Inheritance is sometimes overused and incorrectly used in modelling the relationships among classes.

Aggregations are an alternative way to model these relationships.
3. INCORPORATING COMPOSITION RELATIONSHIPS: EXAMPLE

- Convert inheritance into aggregation.

Diagram:

- 2DArray
- Matrix
- MatrixRepresentation
- output
- 2DArray
- SparseMatrix

Legend:
- Diagonal line: rep (representation)
- Arrow: association
3. INCORPORATING COMPOSITION RELATIONSHIPS

Refactorings regarding aggregations:

Move instance variables/methods from an aggregate class to the class of one of its components.

Move instance variables/methods from a component class to the aggregate classes that contain components which are instances of the component class.

Convert a relationship, modelled using inheritance, into an aggregation and vice versa. [Johnson&Opdyke1993]

Learning objectives:
- Definition and difference between maintenance, evolution, reuse
- Different types of maintenance
- Causes for maintenance and change
- Technique
- Differences of evolution and reuse
POSSIBLE QUESTIONS

▸ Define and explain, in your own words, what an object-oriented application framework is and illustrate it with a concrete example of a framework you know.

▸ Discuss why/how object-oriented application frameworks can achieve software reuse.

▸ Explain, and illustrate with a concrete example, the principle of inversion of control (a.k.a. the Hollywood principle) when building object-oriented application frameworks.

▸ What distinguishes an object-oriented application framework from a library?

▸ What is a hotspot in a framework? Explain and illustrate schematically.

▸ What types of frameworks can be distinguished and what are the main differences between each of these types?
  ▸ (white box / black box / grey box)

▸ Explain and illustrate the Template Method design pattern and discuss its key importance to implement object-oriented application frameworks.
CLASS... IS... DISMISSED.