Chapter 12 Reuse and Pluggable Design — Frameworks in Code

Outline

Code reuse is a much sought-after goal, but it does not happen automatically. It costs money and needs explicit attention, both at the level of design, and in the very structure of the development process within and across projects.

Reuse requires that components be built in a manner which is both generic — not overlied tied to a specific application; and customizable — so it can be adapted to specific needs.

Reuse comes in many flavors, from cut-and-paste, through building libraries of low-level utiltity routines and classes, through skeletons of entire applications with "plug-points" that can be customized. The latter requires a particular mind-set to extracting commonality while deferring just those aspects that are variable.

The key to systematic use of frameworks in code is to make problem descriptions more generic; and to have code techniques for implementing generic or incomplete problem specifications, then specializing and composing them.

12.1 *Reuse and the Development Process*

Software assembly is compelling

One of the most compelling reasons for adopting component-based approaches to development, with or without objects, is the promise of re-use. We would like to build software from existing components primarily by assembling and replacing interoperable parts. These components range from user-interface controls like list-boxes and HTML-browsers, to components for networking or communication, to full blown business objects. The implications on development time and product quality make this very attractive.

12.1.1 What is Reuse?

Many forms of re-use with the same aims

“Re-use” refers to a variety of techniques aimed at getting the most out of the design and implementation work you do. We’d prefer not to re-invent the same old ideas every time we do a new project, but rather to capitalise on that work and deploy it immediately in many new contexts. That way, we can deliver more products in shorter times. Our maintenance costs are reduced too, because an improvement to one piece of design work will enhance all the projects in which it is used. And quality should improve, since re-used components should have been be well tested.

12.1.1.1 Import beats Cut & Paste

Re-use should not just be a one-time thing

Something like 70% of work on the average software design is done after its first installation. That means that any measure aimed at reducing costs must be effective in that ‘maintenance’ phase, not just in the intial design.

Cut-and-paste is one-time re-use

People sometimes think of reuse as meaning cutting chunks out of an existing implementation or design and pasting it in to edit for use in a new one. While this accelerates the initial design process, there is no benefit later on. Any improvements or fixes made to the original component will not propagate to the adopted versions. And if you’re going to adapt a component by such cut-and-paste, you must first look inside it and understand its entire implementation thoroughly — a fine source of bugs!

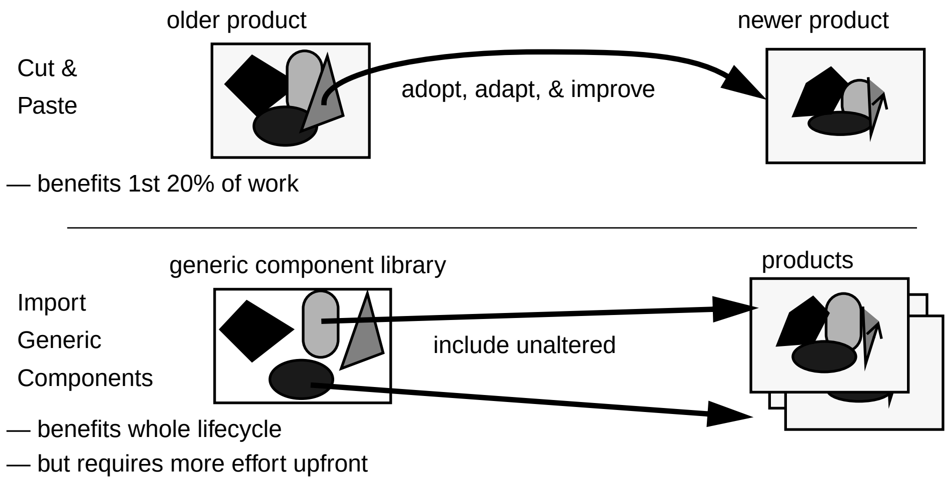

older product
newer product
Cut & Paste
adopt, adapt, & improve
— benefits 1st 20% of work
generic component library
products
Import Generic Components
include unaltered
— benefits whole lifecycle
— but requires more effort upfront

Figure 255: Cut-and-paste Vs. Import

Good reuse therefore implies using the same unaltered component in many contexts, very much like the idea of importing packages described in Chapter 13, *Packages*. Texts on measuring reuse don't count cutting and pasting as proper reuse.

'Import'-based re-use works better

12.1.1.2 The Open-Closed Principle and Reuse Economics

But there is a difficulty. If alterations are not allowed, a component can only be useful in many contexts if it is designed with many parameters and plug-points i.e. if it follows the well-known *Open-Closed* Principle:

But that demands flexible components

Every component should be open to extension, but closed to modification.

It takes effort to work out how to make a component more generic, and the result when deployed might run slower. The investment will be repaid in savings every time the component is used in a design, and every time maintenance only has to be done on one component rather than many slightly different copies. But it is not always certain exactly when or how a component will be reused in the future: so, like all investments, generalising a component is a calculated risk.

Which costs money up front

12.1.2 What are the reusable artefacts?

A reusable artefact is any coherent chunk of work can be used in more than one design. For example:

Many artifacts are reusable

- Compiled code — executable objects
- Source code — classes, methods
- Test fixtures and harnesses
- Designs and models: collaborations, frameworks
- Patterns for analysis and design
- User interface "look & feel"
- Plans, strategies
- Combinations of the above (e.g. spec + code + test harness)

This list includes all kinds of development work. We have already discussed model frameworks and template packages: these can be very valuable to an enterprise. They are "white box" assets: that is, what you see is what is on offer. By contrast, an executable piece of software, delivered without source code, can perform a useful and well-defined function, and yet not be open to internal inspection. Software vendors prefer "black box" components!

We discuss reuse of executable components

12.1.3 A Reuse Culture

In a reuse culture, an organisation focuses on building and enhancing its capital of reusable assets, which would include a mixture of all of these kinds of artifacts. Like any investment, this capital needs to be managed and cultivated. It requires investment in building those assets, suitable development process and roles, and training and incentives that are appropriate for re-use.

Reuse hinges on an 'asset' view of software

To reuse you must generalize your design

But designs have to be generalized to be reusable. Generalising a component can't be justified in terms of its original intended purpose. If you write a collision avoidance routine for your airplanes, there's no reason you should do the extra work to make it usable by your maritime colleagues: you have your own deadlines to meet. And if your product only deals with air traffic, you have no reason to separate out those pieces of the airplane class that could apply to vehicles in general. All these require broadening of requirements beyond what you immediately need to do.

It is easy to generalize for your own reuse

On the other hand, if you notice three lines of code that crop up in six different places in your own product, then you will easily see the point of generalising them and calling a single routine from each place. That's because you're controlling the resources for all the places it could be used; and the problem is small enough to easily get a handle on what's required.

On a larger scale it needs organizational support

On the larger scale, reuse of components between individuals, between design teams, and across and outside organisations takes more coordination. But it's usually someone who holds responsibility for all the usage sites that can assign the resources to get the generalisation done. Reuse needs an organisation and a budget to make it happen.

...and domain modeling

A significant part of identifying large-gained reuse comes from careful modeling of the problem domain itself, and of the supporting domains — UI, communications, etc. — that would be a part of many applications in the problem domain. This activity again crosses specific project boundaries, and also needs organization support.

Think carefully before generalizing

Deciding that a component is to be made sufficiently general for reuse; and how generic and reusable it should be: these are decisions that have to be made consciously and carefully. We have all made such generalizations when deciding that some few lines of code could be moved to a subroutine of their own. Parameterising a whole class or group of classes is the same principle, but employs a wider variety of patterns, and should be more carefully thought through.

How widely will it be used?

Some components can be reused more widely than others. Some objects, routines, or patterns might only be useful in several parts of the same software product: but if it is a big product, or a product family, several teams may need coordinating.

Avoid "galloping generalization"

"Galloping generalisation" is the syndrome where a group spend months producing something that runs like a snail on dope, and has hundreds of interesting features, most of which will never be used. The best strategy seems to be to generalise a component only once it is definitely earmarked for use in more than one context; and then only to generalise it as much as is necessary for the envisaged applications.

Lines of code are a poor productivity metric

Naturally, the organization that measures developer productivity in terms of lines of code written has some re-thinking to do before reuse can succeed. Suitable incentives schemes should be based more on the ratio of code-reused to new code written.

12.1.4 Distinct Development Cycles

It has 2 distinct activities

In a reuse culture, development tends to split into two distinct activities:

- *Product development* — the design and creation of applications to solve a problem. This is centered around understanding the problem, and rapidly locating and assembling reuse capital assets to provide an implementation.

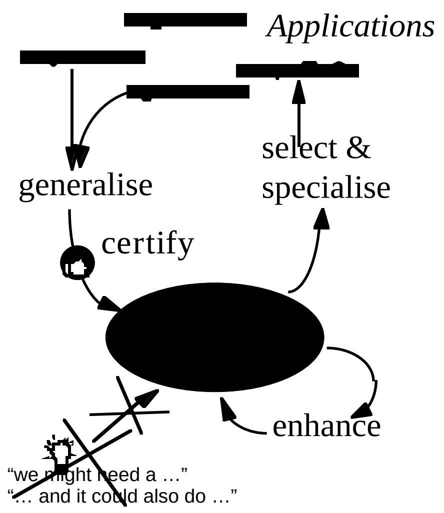
Building products

- *Asset development* — the design and creation of the reusable components that will be used in different contexts. This is carried out with more rigorous documentation and thought. Because software capital assets will be used in many designs, the impact of a change can be, for better or worse, quite large.

and developing assets

It is therefore worth putting much serious effort and skill into assets. Of course, the products must work properly; but whereas much may be gained by, for example, tuning a reusable asset's performance as far as possible, a product which is only used in one context often only needs to be good enough for that purpose. Strong documentation also pays off more with assets than products.

Asset development justifies investment



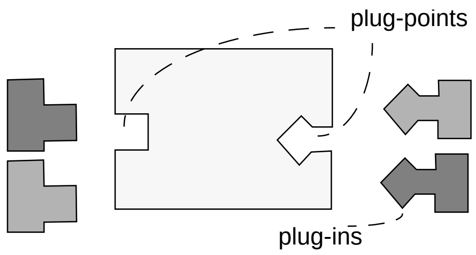
For developing reusable assets that you would generally want to apply many of the techniques in this book. Reuse means investing in the quality of software; the old arguments that "we don't have time to document" can only have a negative effect in a reuse culture. The development of products or applications will also use many of the same techniques, but the process can be quite different (e.g. see Section 11.7, "Heterogenous components," on page 472).

Both use techniques from this book, in different ways

12.2 *Generic Components and Plug-Points*

To make a component that can be reused in different contexts, it must be sufficiently generic to capture the commonality across those contexts; yet it must offer some mechanisms so it can be specialized as needed.

Build *plug-points* for variations



This means you must understand what bits are common across those contexts. Design it so those bits that vary across those contexts are separated from the component itself at carefully selected *plug-points* — places where specialized components can be plugged in to adapt the overall behavior.

Examples

For example, an editor component might work with any suitable spell-checker component that could plug into the interface provided. The spell-checker itself might work with many possible dictionaries, each with its own internal rules about representing and matching its lexicon. Or, a hotel component might use many different room-allocator components that could plug in, each providing different allocation policies via the same interface.

12.2.1 Plugs — the interfaces

Let us look at two kinds of component interfaces

Let's look at a couple of ways in which components can be made to plug together. In particular, it helps to distinguish between composing components to build something bigger, and plugging parts into a generic component in order to specialize its behavior to current needs. Although they both place similar demands on modeling and design, their intent is different.

Upper interfaces — for “normal” use

Components have “upper” interfaces to others they interact with

The components we use are of course made from other components. Figure 256 illustrates how a larger component (a video-rental system) might be assembled by plugging together large components for membership, reservations, and stock management. We think of this as the “upper” interface — it is the direct and visible connecting of parts which provide some well-defined services, without having to adapt or customize them in any significant way. Examples of such “upper interfaces” are the APIs of databases, windows systems, and so on; and, of course, the primary services offered by the membership, reservations, and stock management components.

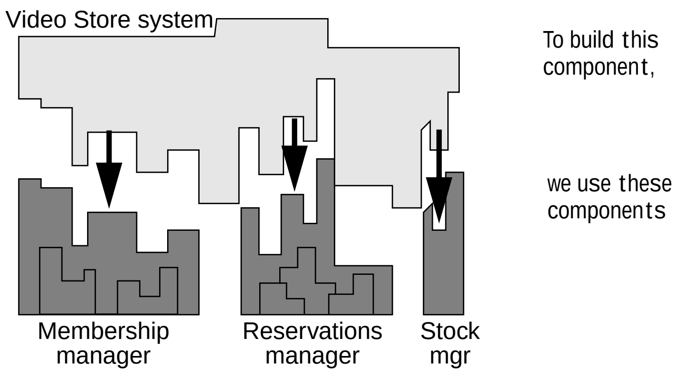
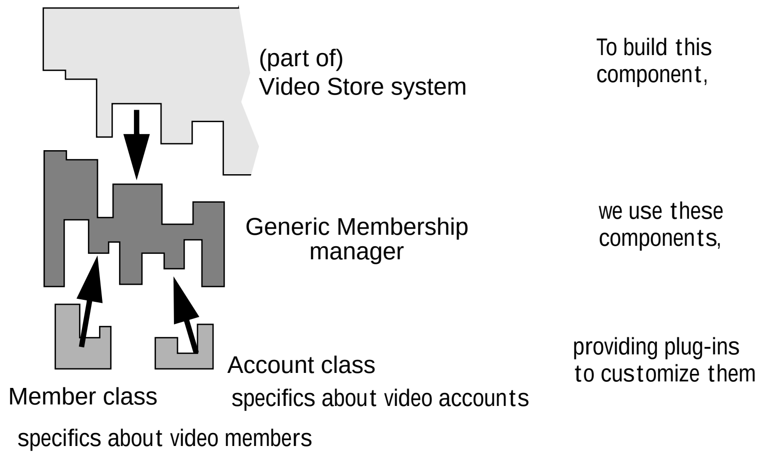


Figure 256: Using upper interfaces to build a larger component

Lower interfaces — for customization

Components often need to be specialized

Each of the components whose “normal” interface we used is itself an incomplete implementation; it needs some additional bits to be plugged into it for it to provide its services. Figure 257 illustrates the use of “lower” interfaces via which a generic component is specialized with several ‘plug-ins’ which customize its behavior to the problem at hand; in this case, a generic membership manager is being adapted by plugging in specifics for video-store members and their accounts.



(part of)
Video Store system
To build this component,
Generic Membership manager
we use these components,
providing plug-ins to customize them
Account class
specifics about video accounts
Member class
specifics about video members

Figure 257: Using lower interfaces to customize a component

Generic components are designed with specialization "plug-points"

Generic components have 'plug-points' — parameterized aspects that can be filled in appropriately in a given context — both for implementation components, as well as for generic model components that are built to be adapted and re-used. When designing a complex component, we might reach into a component repository and build our specific models from generic model components in this library. Of course, the generic versions must provide mechanisms for extension and customization to the specific domain at hand.

Today's software is full of such plug-points

Modern desktop software bristles with plug-points. Web browsers, as well as word-processors and spreadsheets, accept plug-ins for displaying specialized images; desktop publishing software accepts plug-ins for doing specialized image processing. In those cases, the plug-ins generally have to be designed for a particular parent application. The plug-ins are coupled with the parent application when it begins to run, using dynamic linking technologies.

OOP languages support their own plug-ins

Every OO language provides some form of plug-ins. The most common form is the use of "framework" classes: the super-classes implement the skeleton of an application — implementing methods that call operations that must be defined in the subclasses — and a set of subclasses serve to specialize that application. The "plug-points" are the subclasses and their overriding methods. In C++, a template List class can be instantiated to provide lists of numbers or lists of elephants, or of whatever class the client needs; the "plug-point" is a simple template parameter.

Frameworks of multiple classes also constitute generic components

The principle is not limited to single classes, but can span multiple abstract classes that collaborate with each other, and have to jointly extended before use. Many class libraries provide user interface frameworks (like Smalltalk's Model-View-Controller). You make a user interface (either manually or with a visual builder tool) by inheritance from the framework classes and plugging-in specializing methods into your subclasses.

12.2.1.1 Infrastructure Services: a special kind of "lower" interface

Common services, not customized plug-ins

Many components need some underlying set of infrastructure services to be provided to operate properly (Section 11.1.4, "Components and Standardization," on page 443). These do not customize the behavior of the component in any interesting way; they simply provide an implementation of a common "virtual machine" for use by all components.

Obvious examples of this include the POSIX interface that provides a common view of many different operating systems; and the Java virtual machine, which provides all the services needed to run Java components. However, this underlying virtual machine may be more specialized to the problem at hand e.g. a state-machine interpreter, or a graph transformation engine.

12.3 *The “Framework” approach to code reuse*

In object-oriented design and programming, the concept of a “framework” has proven to be a very useful way to re-use large-grained units of design and code, while permitting customization to different contexts. The style of reuse with frameworks, and mind-set for factoring out commonality and differences, is quite distinctive.

Frameworks — large grained flexible reuse

12.3.1 OOP frameworks

An object-oriented framework is often characterized as a set of abstract and concrete classes that collaborate to provide the skeleton of an implementation for an application. A common aspect of such frameworks is that they are adaptable i.e. the framework provides mechanisms by which it can be extended, such as by composing selected sub-classes together in custom ways, or defining new sub-classes and implementing methods that either plug-into or override methods on the superclasses.

Traditionally, a framework is a collection of collaborating adaptable classes

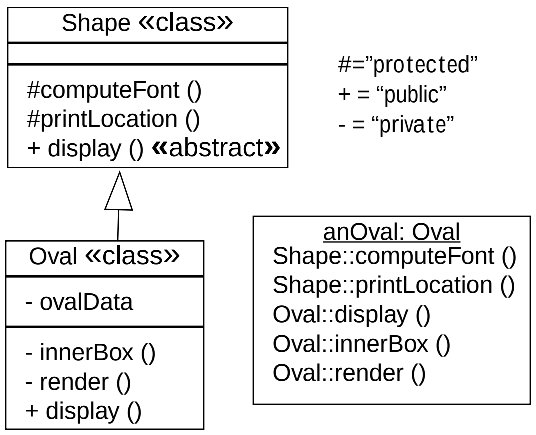
There are some fundamental differences between the framework style and more traditional styles of re-use, as illustrated by the example below.

A framework has important differences from more traditional re-use forms

> *Design and implement a program for manipulating shapes. Different shapes are displayed differently. When a shape is displayed it show a rendering of its outline and a textual printout of its current location in the largest font that will fit within that shape.*

In the next two sections we will contrast the traditional approach to reuse with the framework style of factoring for reuse.

Class-library with “traditional” reuse



A “traditional” approach to re-use might factor the design as follows. Since the display of different shapes varies across the kind of shapes, we design a shape hierarchy. The *display* method is abstract on the superclass — since shapes display themselves differently — and each subclass provides its own implementation.

Main method is “abstract” on Shape

There are some common pieces to the display method e.g. computing the font size appropriate for a particular shape given its “inner” bounding box, and printing out the location in the computed font. Hence we implement a *computeFont* and *printLocation* method on the superclass (marked ‘protected’ in Java, to be subclass-visible).

Common bits are implemented on Shape

```
class Shape {
    // called from subclass: given a Bounding Box and String, compute the font
```

```
        protected Font computeFont (BoundingBox b, String s) { .... }
        // called from subclass: print location on surface with Font
        protected void printLocation (GraphicsContext g, Font f, Point location) { ... }
        public abstract void display (GraphicsContext g);
    }
```

Subclass calls inherited common bits

A typical subclass would now look like this:

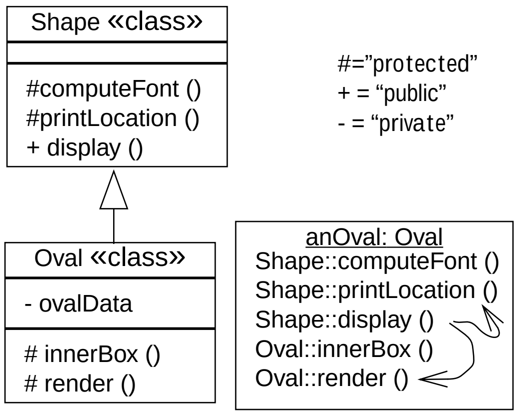
```
    class Oval extends Shape {
        // shape-specific private data
        private LocationInfo ovalData;
        // how to compute my innerBox from shape-specific data
        private BoundingBox innerBox() { .... }
        // rendering an oval
        private void render (GraphicsContext g) { ... trace an oval ... }
        // display myself
        public void display (GraphicsContext surface) {
            render ();
            BoundingBox box = innerBox();
            // let the superclass compute the font
            Font font = super.computeFont (box, ovalData.location.asString());
            // let the superclass print the location
            super.printLocation (surface, font, ovalData.location);
        }
    }
```

Framework-style re-use and “Template method”

An inversion

With framework development, the skeleton of the common behavior is a 'template method' in the superclass; the variant bits and pieces are deferred to the subclasses.

Frameworks — seek variations to defer



With a “framework” approach to re-use our factoring looks quite different. We start off with the assumption that all shapes fundamentally do the same thing when they are displayed: render, compute a font for their inner bounding box, and print their location in that font. Thus, we implement at the level of the *superclass*:

Skeletal code in superclass

```
    class Shape {
        public void display (GraphicsContext surface) {
            // delegate to subclass to fill in the pieces
            render (surface);               // plug-point: deferred to subclass
            BoundingBox box = innerBox(); // plug-point: deferred to subclass
            Point location = location();    // plug-point: deferred to subclass
```

```
                // then do the rest of "display" based on those bits
                Font font = computeFont (box, location);
                surface.printLocation (location, font);
            }
    }
```

The actual rendering and computation of the inner box and location must be deferred to the subclasses — *plug-points*. However, *if* a subclass provided the appropriate bits as *plug-ins*, it could inherit and use the same implementation of *display*. Thus, we are imposing a consistent skeletal behavior on all subclasses, but permitting each one to flesh out that skeleton in its own ways.

Defer variation points to subclass plug-ins

```
    class Oval extends Shape {
        // implement 3 shape-specific "plug-ins" for the plug-points in Shape
        protected void render (GraphicsContext g) { ...trace an oval ...}
        protected BoundingBox innerBox () { ... }
        protected Point location () { return center; }

        // private shape-specific data
        private Point center;
        private int majorAxis, minorAxis, angle;
    }
```

Although this example focuses on a single class hierarchy, it extends to the set of collaborating abstract classes that are characteristic of frameworks. The Shape hierarchy, for example, requires certain services from the GraphicsContext object to display itself; there could be different implementations of GraphicContext as well, for screens and printers, using a similar framework-styled design. It is this partitioning of responsibility — between different shape classes, and between shapes and the GraphicsContext — that gives the design its flexibility. Thus, any packaging of a class as a re-usable unit *must* also include some description of the behaviors expected of other objects i.e. their *types*.

Extends across multiple collaborating classes

Contrast of styles

Note the contrast between the approaches, in terms of factoring of code, degree of re-use, and consistency of resulting designs.

A different mind-set

Traditional	**Framework**
Begin with the mind-set that the display methods would be *different*, and then seek the *common pieces* that could be *shared* between them	Assert that the display methods are really the *same*, and then identify the *essential differences* between them to *defer* to the sub-classes
Focus on sharing the *lower-level* operations like computeFont and printLocation. The higher-level application logic is duplicated, and each one calls the shared lower-level bits.	Share the *entire skeleton* of the application logic itself. Each application plugs into the skeleton the pieces (like render, innerBox, particular GraphicsContexts) required to complete the skeleton

Traditional	Framework
Most calls go from the application to the shared base	Most (many) calls go from the framework skeleton to the individual "applications"; in fact, one of the hallmarks of a framework is *"don't call me — i'll call you"*
Define an interface that the applications can use to call the reusable parts	Define an interface representing demands that the reusable skeleton framework makes on the applications — the plug points for extension
The application contains the newer code; the base contains old code; newer code calls existing code	Application contains newer code; however, the existing (base) code calls newer code to delegate specialized bits.

These differences are summarized in Figure 258, showing the contrast between base and application levels in the two approaches.

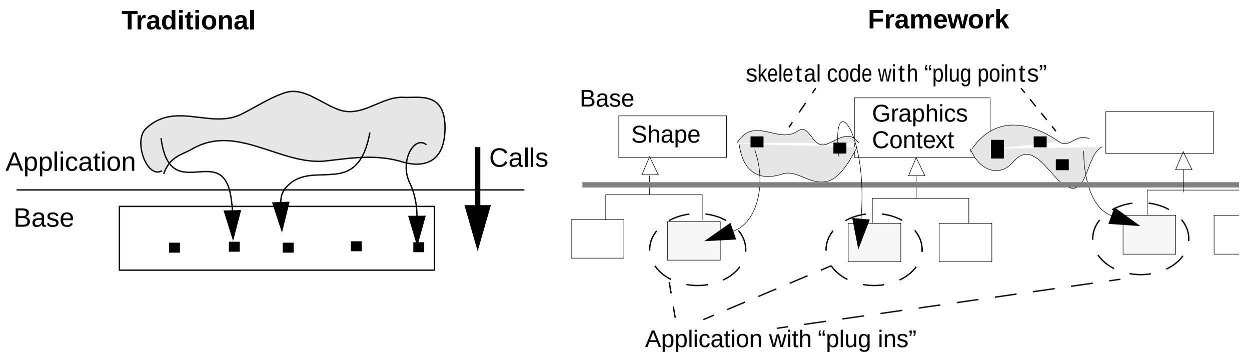


Figure 258: Traditional vs. framework-style designs

Frameworks provide *plug-points* as interfaces for customization

A significant part of framework design is factoring the *plug-points* that are provided for adaptation or customization. This example requires that a subclass provide the missing behaviors. Other frequently recommended design styles for frameworks are based on delegation and composition; by composing an instance of a framework class with an instance of a custom class that implements a standard framework interface, we adapt the behavior of the framework.

12.3.2 Non-OOP Frameworks

Object-oriented programming provides some novel ways to implement with plug-points and plug-ins. However, the underlying style of implementing a skeletal application, while leaving places in it that can be customized, can be achieved with other techniques as well, the most central of which is delegation (Section 12.5.3, "Polymorphism and Delegation," on page 512).

12.4 *Frameworks — Specs to Code*

In the previous section we saw how framework techniques in object-oriented programming can help build a skeletal implementation, with plug-points for customization to specific needs. We saw in Chapter 10, *Model Frameworks and Template Packages* , that pieces of code are not the only useful re-usable artifacts; recurrent patterns occur in models, specifications, collaborations. Moreover, the basic OOP unit of encapsulation — a *class* — is not the most interesting unit of describing designs; it is the collaborations and relationships between elements that constitutes the essence of any design.

12.4.1 Generalize/Specialize: models and code

We want to apply framework-like approaches throughout the development lifecycle

In order to systematically apply framework-based techniques to development, we start with template packages to construct domain models, requirements specifications, and designs from frameworks. The specifications for a particular problem could be constructed by applying the generic framework and "plugging" in details for the problem at hand. On the implementation side, an implementation for the generic specification should be correspondingly customizable for the specialized problem specification (Figure 259).

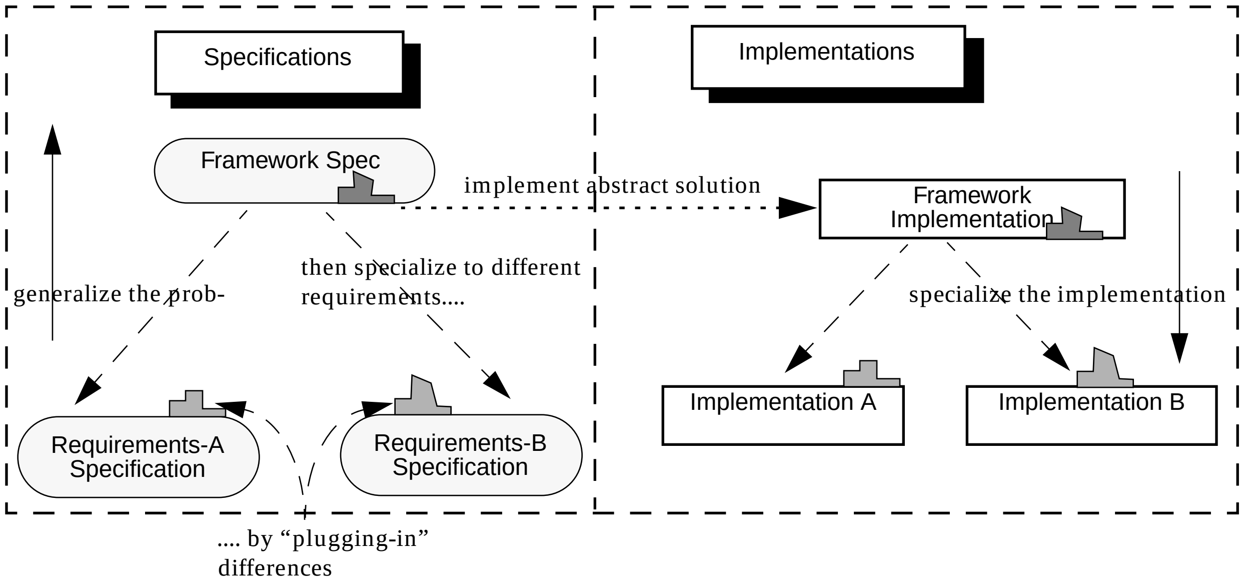

Specifications
Implementations
Framework Spec
implement abstract solution
Framework Implementation
generalize the prob-
then specialize to different requirements....
specialize the implementation
Requirements-A Specification
Requirements-B Specification
Implementation A
Implementation B
.... by "plugging-in" differences

Figure 259: Frameworks for Specification vs. Implementation

We can implement a solution to an abstract problem

A framework implementation thus provides a customizable solution to an abstract problem. If done right, the points of variability in the problem specifications — *plug-points* on the specification side — will have corresponding *plug-points* on the implementation side as well[1].

1. The relation between these plug points is analogous to refinement

12.4.1.1 Combining Model Frameworks

Example of a seminar company

Consider the operations of a service company that markets and delivers seminars. Different aspects of this business, and hence its software requirements, can be described separately: allocation of instructors and facilities to a seminar, on-time production of seminar materials for delivery, trend analysis for targeted marketing of seminars, invoicing and accounts-receivable, etc.

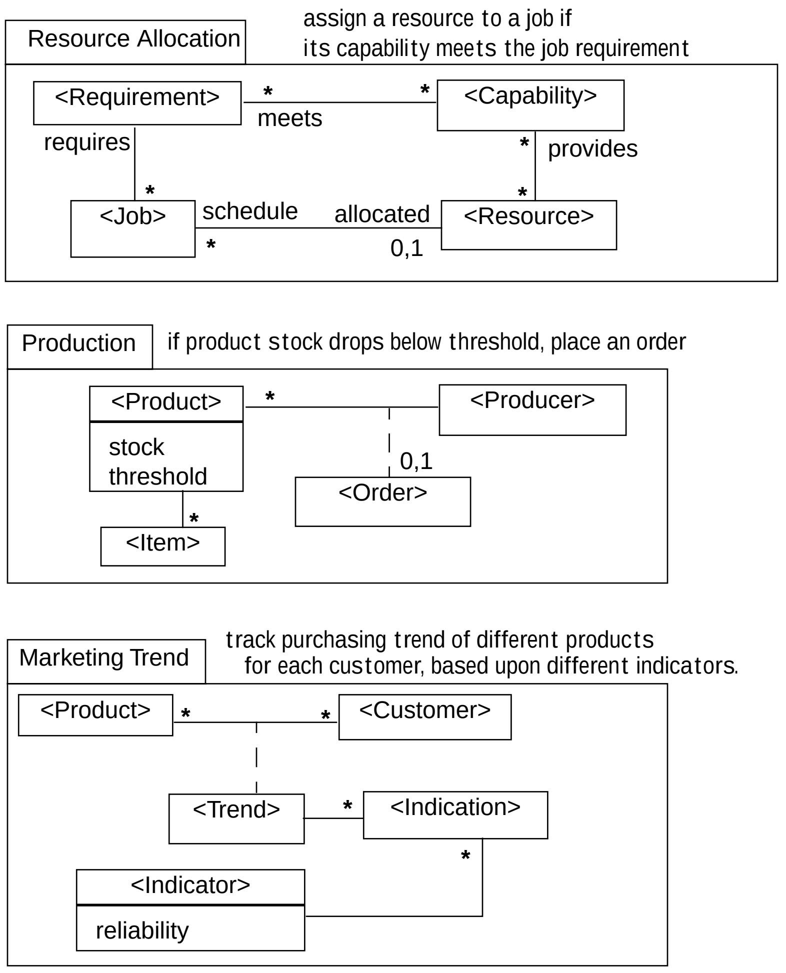


Figure 260: Specification frameworks for the seminar business

Each such aspect can be generalized to be independent of seminar specifics, creating a library of re-usable abstract specification frameworks, shown in Figure 260 with details of invariants and action specs elided.

This framework uses abstract types like *Job, Requirement,* and *Resource,* and abstract relationships like *meets, provides,* etc. These will map in very different ways to a car rental application (*Resource=Vehicle, Job=Rental, meets=model category matches*) than to assigning instructors to seminars (*Resource=Instructor, Job=Session, meets=instructor qualified for session topic*).

These frameworks must now be mapped to our problem domain, and related to each other by shared objects, attributes, etc. Figure 261 shows the overall problem model as an application of these frameworks.

It can be modeled by framework composition

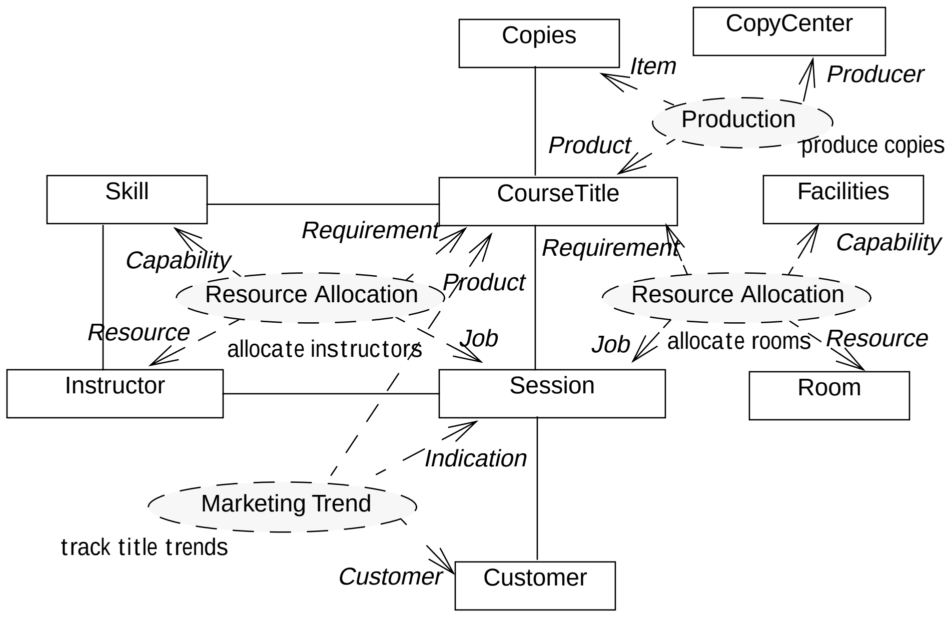


Figure 261: Specification by composing frameworks

Of course, these frameworks must interact with each other. A session must have both an instructor and room assigned; failure of either means the session cannot hold. When a session holds, copies of course materials must be produced, and the customer trends get updated. Note that each problem domain object can play multiple roles in different frameworks. For example, a Course Title serves as a *Requirement* in the two applications of the *Resource Allocation* framework, and as a *Product* in the applications of the *Marketing* and *Production* framework.

The composite specifies framework interactions

12.4.1.2 Combining Code Frameworks

Each of the model frameworks in Figure 261 could come with a default implementation frameworks. Our design, at the level of framework-sized components, would look like Figure 262:

Each model framework could have an implementation

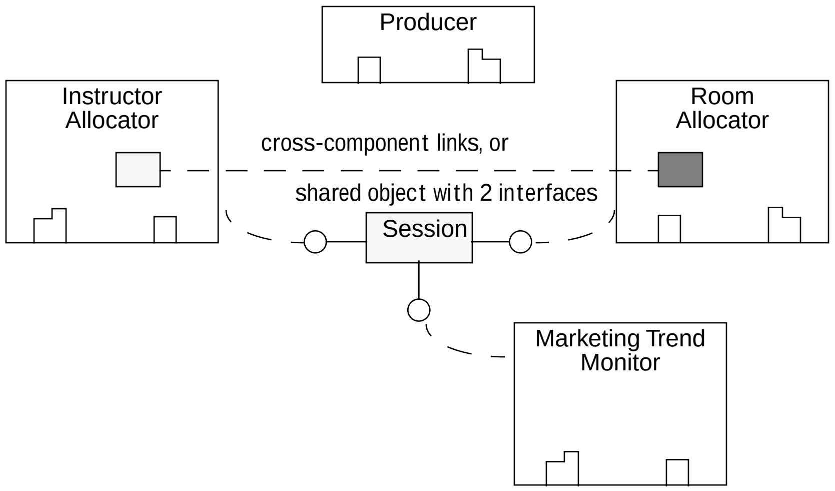


Figure 262: Customizing and connecting implementation frameworks

With suitable plug-ins

Each of the components has its plug-points suitably filled by implementation units from this problem domain. Thus, the instructor allocator has Instructor and Session as plug-ins for Resource and Job; and the trend watcher has Session and Topic plugged in for Indication and Product.

Here are two (of many) schemes to make these frameworks interact with each other:

Can share objects across roles

- A shared object for the session, offering different interfaces for each role it plays. Its class will override some framework methods to make explicit invocations into the other frameworks, if necessary.

```
// one interrface for each role this shared object plays
class Session implements Indication, Job, ... {
    // just became confirmed from the allocation framework
    void confirm () {
        ...do normal confirmation stuff
        // but confirmation must update the marketing indicators
        MarketingTrendMonitor::indicationConfirmed (self);
    }
}
```

Or use cross-component links

- Separate objects for each role in each framework, with some form of cross-component links between them. These objects can inherit a default implementation.

```
class SessionJob implements Job extends DefaultJob {
    .// link to the corresponding “Indication” object
    Indication trendIndication;

    // just became confirmed from the allocation framework
    void confirm () {
        super.confirm () // do normal confirmation stuff
        // but confirmation must update the marketing indicators: directly, ...
        MarketingTrendMonitor::indicationConfirmed (trendIndication);
        // or, delegate to the indication object
```

```
            trendIndication.confirm ();
        }
    }
```

A third way is to uniformly compose roles (Pattern-8, *Role delegation* (p.514)).

12.5 *Basic Plug Technology*

There are several implementation mechanisms for achieving the effect of plug-points and plug-ins. This section discusses the main ones.

12.5.1 Templates

Type-parameterized class family

C++ provides a compile-time template facility which can be used to build generic classes, or families of generic classes. One way to implement a framework for resource allocation woudl be to use a family of C++ template classes that are mutually parameterized:

```
template <class Job>
class Resource {
    Set<Job*> schedule;
    makeUnavailable (Date d) {
        ...
        for ( the job in schedule overlapping d, if any )
            job.unconfirm ();
    }
}

template <class Resource>
class Job {
    Resource* assignedTo;
    Range<Date> when;
    unconfirm () { .... }
}

template <class Resource, class Job>
class ResourceAllocator {
    Set<Resource*> resources;
    Calendar<Resource*, Job*> bookings;
    ...
}
```

With inheritance for roles

We can use inheritance to have an instructor get the resource behavior for a session:

```
class Instructor :public Resource<Session*>, ...
```

We might use multiple-inheritance[1] to have our Session play the role of Job for 2 resources:

```
class Session :public Job<Instructor*>, public Job<Room*> {
    ....
}
```

1. Some circularities in type dependencies will not work with C++ templates.

12.5.2 Inheritance and the Template Method

For an inheritance-based design, the template method (Section , “Framework-style re-use and “Template method”,” on page 502) forms the basis of plug-ins. This was a greatly overused design style initially, and has now fallen out of favor.

12.5.2.1 Inheritance is just one narrow form of reuse

Inheritance was initially touted as the preferred object-oriented way to achieve re-use and flexibility. In the early days of Smalltalk (one of the earliest popular OO programming languages), several papers were written promoting “programming by adaptation”. The principle was that you take someone else’s code, make a subclass of it, and override whichever methods you require to work differently. Given, for example, a class that implements Invoices, you could define a subclass to implement BankAccounts: they both are lists of figures with a total at the end.

Inheritance with free overriding was touted as the key to flexibiilty

Although the code runs OK, this wouldn’t be considered good design. The crunch comes when your users want to update their notion of what an Invoice is. Because a BankAccount is a different thing, it’s unlikely that they’ll want to change that at the same time, or in the same way; or that the overrides retain the behavior expected of an Invoice. It then takes more effort to separate the two pieces of code after the change, losing whatever savings there were in the first place.

This caused problems with code revisions

The programmer who uses inheritance like this has forgotten the cut and paste keys: they provide the proper way to start a design that takes over some ideas from another one. If the concepts are unrelated, then the code should be as well.

Do not inherit, or re-use, code unless you want its spec as well

Do not re-use code unless you also intend to re-use its specification, since the internal implementation itself is always subject to change without notice.

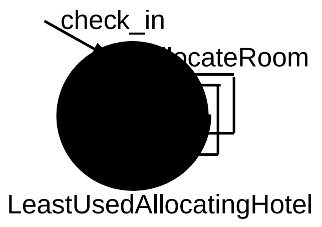
12.5.2.2 Inheritance does not scale for multiple variants

So what else might inheritance be good for? Perhaps multiple variants of a basic class. Take for example a hotel booking system. When a guest checks in, the system does various operations, including allocating a room. Now, different hotels allocate their rooms with different strategies: some always choose the free room nearest the front desk; some allocate circularly to ensure no room is used more than another; and so on.

Perhaps inheritance can be used to handle variants

So we have several subclasses of Hotel, one for each room-allocation strategy. Each subclass overrides allocateRoom( ) in its own way. The main checking-in function delegates to the subclass.

```
class Hotel
{    public void check_in (Guest g)
     { ... this.allocateRoom (g); ...}
     protected abstract
          Room allocateRoom (Guest g);
}
class LeastUsedAllocatingHotel extends Hotel
{
     public Room allocateRoom (Guest g) {....}
```



But how to combine different variant features?

But of course the problem is that it is difficult to apply this pattern more than once: if Hotels can have different staff-paying policies, does that mean we must have a different subclass for each combination of room-allocation and staff-payment? That will not scale very well, even if you did have multiple-inheritance.

12.5.3 Polymorphism and Delegation

Polymorphic delegation works much better

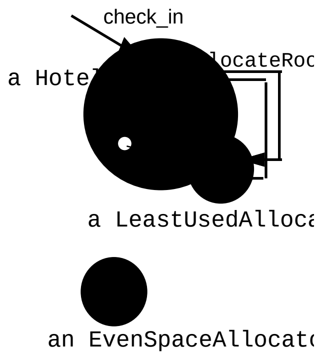
The solution is to delegate these tasks to separate specialist ‘strategy’ objects that implement different policies behind a common interface [Gamma, *State* p305]; this is the essence of good polymorphic design.

```
class Hotel {
    Allocator allocator;
    public void checkInGuest (Guest g)
    {... allocator.doAllocation(g);..}
}

class Allocator {
    Room doAllocation (...); // returns a free room
}

class LeastUsedAllocator implements Allocator {
    Room doAllocation (...) {...code ...}
}

class EvenSpaceAllocator implements Allocator {
    Room doAllocation (...) {...code ...}
}
```



Each Hotel object is coupled to a room-allocator object, to which it delegates decisions about allocating rooms. Separately, it is coupled to a staff-payer, and the same for whatever other variant policies there may be. Different policies are implemented by different classes, which may be completely different in their internal structure. The only requirement is that all room-allocator classes must implement the doAllocation( ) message: that is, they must conform to a single interface specification.

You can assemble many combinations from few parts

This polymorphic coupling between objects is far more important as a design principle than inheritance. It is what enables us to link one component to many others, and thereby to build a great variety of systems from a well-chosen set of components. Both component-based and more ‘pure’ objects-oriented approaches can take good advantage of this delegation-based approach via interfaces.

Going back to the BankAccount and Invoice example: if there is really any common aspect to the two things, the proper approach is to separate it into a class of its own. A list of figures that can be added up might be the answer; so while BankAccount and Invoice are separate classes, they may both use ListOfFigures.

12.5.4 Good uses for inheritance

There are some good uses, and many bad ones

So is there any good use for inheritance? Extremists would say we can do without it, and write very good object-oriented software, provided we have the means (a) to document and check interface implementation, and (b) to delegate efficiently to another object without writing too much explicit 'forwarding' code. All object-oriented programming languages support these techniques, though some do so better than others. Java, for example, has good support for interfaces, while C++ confuses implementation and inheritance. Smalltalk has support for inheritance, but no type-checking. Languages properly supporting delegation are few: it can be done in Smalltalk, and Java gets half-way there with its inner classes. Perhaps the next fashionable successor to Java will have explicit support for delegation.

More pragmatically, class inheritance does have its place and value; but it should not be used where delegation via a polymorphic interface would work. Inheriting from an abstract class, which provides an incomplete or skeletal implementation, and then extending it to plug-in bits specific to your need, is reasonable; inheritance with arbitrary overriding of methods is not advisable.

12.5.5 A Good Combination

Layer interface, abstract class, and concrete class

One good way to combine these techniques is as follows:

- For every role define an interface:

  ```
  interface IResource { .... }
  ```

- For every interface, define a default implementation with inheritance plug-points:

  ```
  abstract class CResource implements IResource {
      protected abstract plugIn ();
      public m () { ..... plugIn(); ...}
  }
  ```

- Each default implementation should itself delegate to other *interfaces*:

  ```
  abstract class CResource implements IResource {
      private IJob myJob;
  }
  ```

- Concrete classes will typically inherit from the default implementation; but they could also independently implement the required interface.
- Use a factory to localize the creation of new objects of the appropriate subclasses:

  ```
  class ResourceFactory {
      IResource newResource () {
          return new CResource;
      }
  }
  ```

  That way you can make a local change to the factory and have entirely new kinds of resources be created and used polymorphically.

Pattern-8 Role delegation

Summary Adopt a uniform implementation architecture based on composition of separate 'role' objects, to allow plugging together of code components.

Intent To compose separately implemented objects for different roles.

Objects play several roles, each of which may have several variants. We don't want a separate class to implement every combination of all the variants e.g. a Person can be a Full-time or a Part-time Employee; a Natural, Foster, or Step-Parent; and so on. The set of roles (and the choice of variant) may change at run time. We need to change the type without losing the object's identity.

Combining two specifications is easy: you just AND them together (that is, you tell the designer to observe both sets of requirements). You can't do that with code, so we look for a standard mechanism for cooking up an object by systematically combining roles from several collaborations. This will enable us to stick with the big idea that design units are often collaborations (not objects), but still have the convenience of plugging implemented pieces together like dominoes.

Srategy The technique is to delegate each of the role-specific pieces of behaviour to a separate object. One conceptual object is then implemented by several: one for each role, and (usually) a 'principal' to hold them all together. The principal object keeps those parts of the state to which access is shared between the roles. Each role conducts all dialogue with the other participants in the collaboration from which it arises. Generally, the roles are designed as observers of various pieces of the principal's state.

Make the group behave to the outside world as a single object, which was the original intent, by always keeping them in sync and being careful with 'identity' checks.

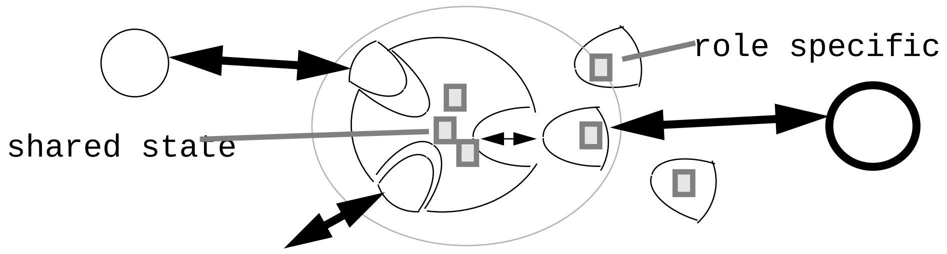


Figure 263: Building objects by connecting "role" objects

You will need to design an interface for all plugins to the same principal, so new plug-ins can be added for new roles. Never use a language-defined 'identity-check' (== in Java); instead, have a 'sameAs(x)' query — plug-ins pretend they're all same object if they share the same principal. Calls to 'self' within plugins usually go to principal.

For example, the basic trading principal has a stock of products and cash assets. Into this can be plugged a role for retailing, that knows about a Distributor and monitors the stock level, generating orders when necessary. Or we could make it a Distributor, plugging in the appropriate role — perhaps a Dealer would be something with both the Retailer and Distributor roles.

Pattern-9 Pluggable roles

Summary

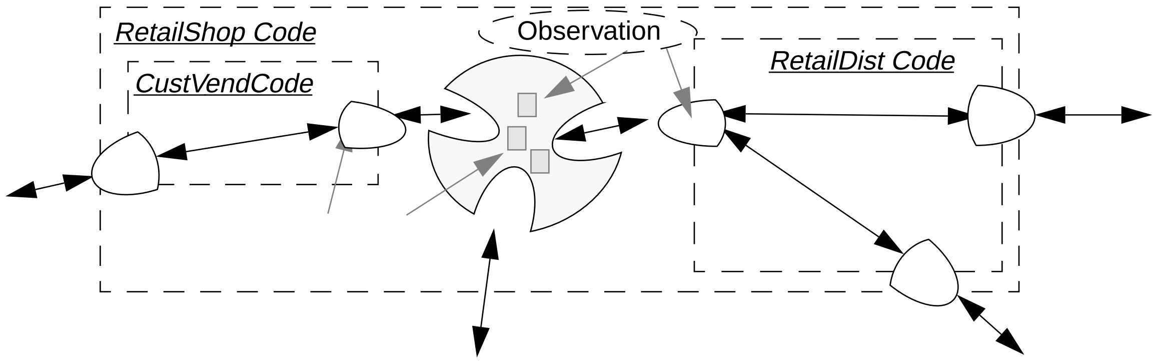
Make role objects share state via observation of a shared object.

Intent

We need to supply complete implementations of frameworks, but frameworks are often about collaborations between roles, rather than complete object behaviors.

Strategy

- Implement components as collaborations between role plug-ins
- Each role implements the responsibilities of its framework spec
- Each role is an Observer of the shared state



- Ensure a common interface for plug-ins
- Designers couple principals to collaborations, to build new collaborations

Roles observe shared state. So that a fully-coded component can mimic the structure of the corresponding specification frameworks, each role should incorporate the code necessary for implementing placeholder actions. Most placeholder triggers boil down to monitoring changes of state. Each role can therefore be built as an observer of the parts of the common state that it is interested in.

The principal provides a standard pluggable interface allowing each role to register its interests, and makes each sharable attribute a potential subject.

Collab components mirror framework specs. After building a specification by composing framework models, you can implement it by plugging together the corresponding fully-implemented collaborations (if they are available).

This scheme could pay some performance penalty compared to purpose-built systems. There is overhead in the wiring of the observers wherever components are plugged together; although more efficient versions of Observation, such as the JavaBeans event model, may adequately address this. In exchange for performance, you get rapid development; and you always have the option of designing an optimised version, working from the composed framework specifications.