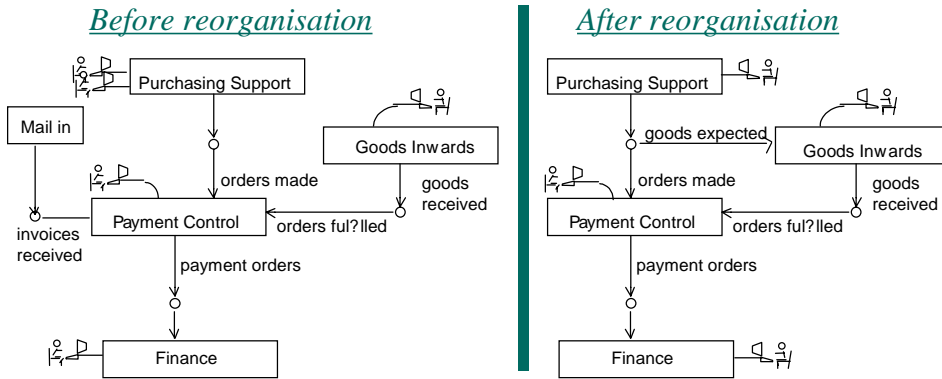


Component Based Development

Alan Cameron Wills
TriReme International Ltd
<http://www.trireme.com>



This could be an organisation chart for a business, showing an improvement in the flow of work. (It's based on a real example.) But it could also be a software organisation chart: for if each department is supported by appropriate software, then the lines of communication in the business will be mirrored by communication between the supporting software components. And when the business is restructured, of course the software must be restructured at the same time.

In most businesses, the software is the brake on business reorganisation: it takes a long time to make such restructurings. Component Based Development promises to make reorganising the software just as easy as reorganising the business.

Component Based Development (CBD) is one of the more recent in a long history of phrases to buzz around the world of software development. Like its predecessors, it represents the desire for better software delivered more cheaply and reliably; like them, the hype will exceed the actual benefits; and like the others, we shall learn something from it that is absorbed into our gradually improving engineering discipline.



So what is CBD about? What are its potential benefits, and what do you need to do to get them? This article is an introduction.

Contents

1. [What does business need from software?](#) Flexibility as well as functionality.
2. [How not to make flexible software](#) Poorly-decoupled code, copied and modified.
3. [How to make flexible software](#) Decoupled 'plug-in' components.
4. [Where do we see components ?](#) From widgets to enterprise-level systems.
5. [Components, kits, and connectors](#) Families of products from kits of components.
6. [Component Kit Architecture](#) Defining how the components couple together.
7. [The common model](#) Each component has its own view of the business.
8. [Wiring](#) The basic technology.
9. [Reusable assets](#) Managing the development of components.
10. [Components compared with ...](#) Is a component a module? Or an Object ?
11. [Summary](#) Decoupling on any scale gives flexible design.
12. [Further reading](#)

1 What does business need from software?

Software supports business (whether directly by supporting business activities, or secondarily by driving machinery that you use or make). So what should software designers provide for the business?

Supportive Function	What the software does should further the business goals. “Correct function relative to requirements” might be a more ideal heading here; but in general, part of the project is to establish the requirements by analysing the business goals and operational practices.
Quality	Secure — shouldn’t threaten life or business. Robust — shouldn’t fall over. Correct — what is provided should work according to the established spec. Complete — a sensible subset of the original vision (can we hope for more?)
Flexibility	Changes. The business requirements change over time. <ul style="list-style-type: none">• E.g. a finance company wants to offer a new service feature to customers, to keep ahead of the competition. The software needs to provide for the new deal.• E.g. a mobile phone manufacturer wants to provide a new feature, to keep up ahead of the competition. The embedded software must provide the new feature.• E.g. a web browser vendor needs to keep customers’ attention by continual introduction of new features. Variants. The business requirements vary between different installations. <ul style="list-style-type: none">• E.g. the finance company must observe local laws in different countries.• E.g. the mobile phone company makes several models of phone. E.g. some browser users want to download music; others want to send email.
On time, in budget	Some hope! Still, we can try to predict time & cost more accurately, and we can shorten times by reuse — whatever that means (see below).

Traditional methods of software development have to a large extent cracked the challenges of Supportive Function and Quality: most of us can write a program that works — though we can’t claim that these methods are practised everywhere!

Flexibility, by contrast, is something that is still very variable. We are not generally well practised in delivering it. We never get the opportunity to try our hands at it, because it is often one of the first goals to be dropped as deadlines approach. Partly, this is because flexibility is quite difficult to measure: the customer can test for Supportive Function and Quality, and generally seem aware of deadlines and budget; but if Flexibility hasn’t been delivered, they don’t know about it for some time.

2 *How not to make flexible software*

In many software establishments, there is no clear agreement about how to provide for changes and variants. Typically:

No documents

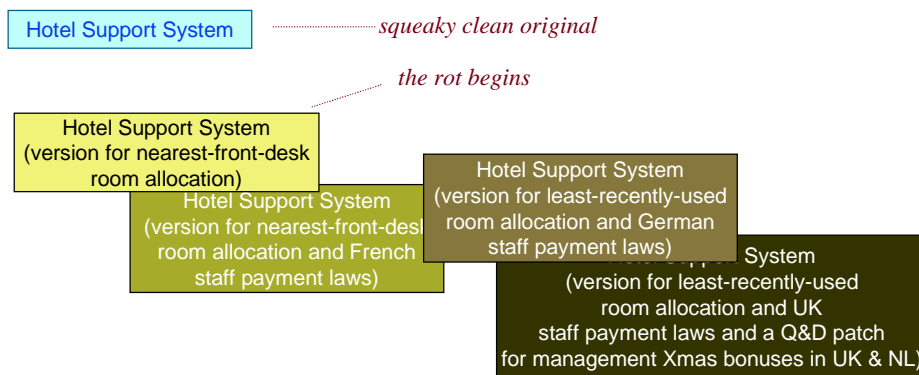
As staff come and go, the original coherent vision of the design is lost. Called upon to make a change, maintainers do not have time to understand how to make the change fit perfectly. The design degrades over time.

Coupling

Any one requirement is realised in several parts of the design, and each part mingles several requirements. Every update propagates throughout the design; or more likely, maintenance deadlines mean that restrictions and dependencies are introduced, so that eventually it becomes difficult to tweak one end without the other falling over.

Local copies

Several versions of the software are developed to cope with variants. These gradually diverge, needing separate maintenance teams, and becoming different designs. When the need arises to make the different variants talk to each other, they can't.



For example, a hotel chain originally plans to install the same software in every hotel. But with expansion, new managers express different preferences as to how to allocate guests to rooms. The data structures required are different; and room availability checks, guest arrival, transfers, and check-out are all dealt with in different parts of the design. This results in two or more versions of the complete software.

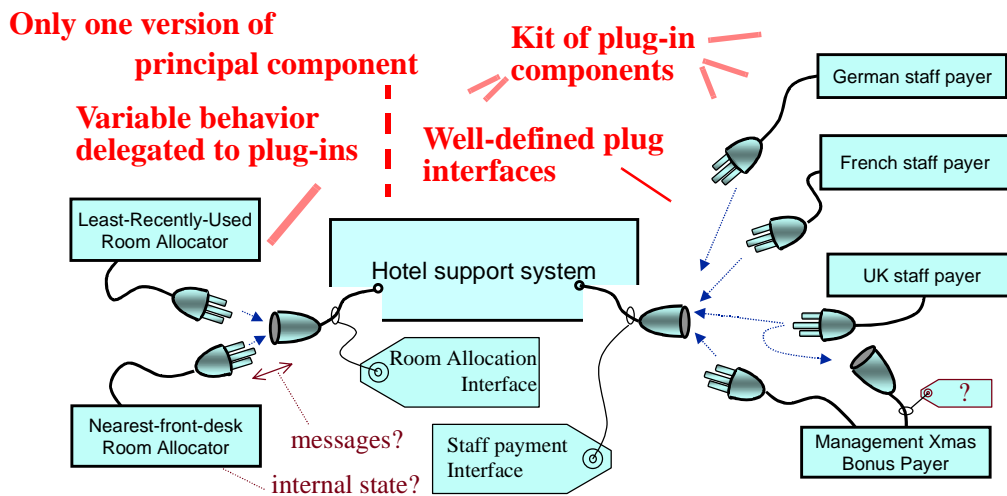
Then the chain expands to different countries, in which different regulations apply. The different countries' head offices hire their own programmers to adapt the software to local requirements. It soon becomes difficult to apply an improvement made in one version to any of the others: and they become effectively different programs.

After a few years of successful operation, the chain wants to provide international reservation facilities on the internet, and to do deals with international corporations. But it finds that the software in its different outposts cannot communicate with each other. In fact, neither can the staff, because their business practices and concepts have diverged: when Hans talks about a Customer, he means the person who stays in his Hotel; François means the person who pays the bill; Tom means anyone on their mailing list.

3 How to make flexible software

Build with pluggable components

- Separate each variable piece of behavior into separate “plug-in” components: create different components to deal with different variants. Define a common interface so that different variants can easily be substituted.
- Make a kit of components for building systems within your business.
- Assemble variants of the basic system only by configuring the components in different ways. Do not permit local rewrites of components.
- Ideally, plan all the whole family of end-products, with the component-kit designed to provide for all the envisaged variants.
- The plug-in components, and the ‘plugging’ mechanism can take various forms: tables, parameters, object-oriented polymorphism, The principle is the same.



In the example, there are several independently variable requirements, including how to allocate guests to rooms and the way that staff are paid. In the plug-in solution, each variable requirement corresponds to a ‘socket’: once that interface is defined, you can design a plug-in part for each variation in that requirement.

Precedents

Hardware designers have been doing this for years. Your CD player will work with any make of speaker; your fax will talk to mine; the design of your car shares most of its components with other models made by the same or different manufacturers.

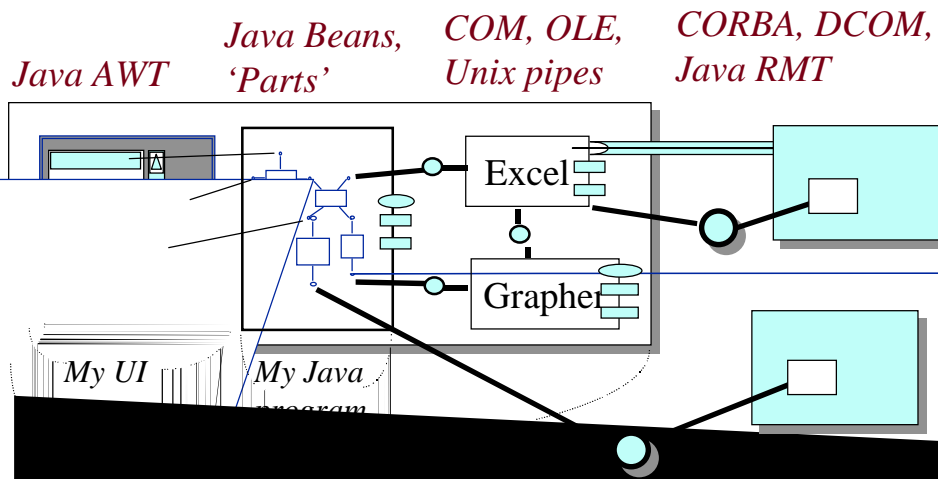
Modern desktop software (for example, web browsers) is built on the pattern of big central component + third-party plug-ins.

Operating systems and databases have for many years worked with many applications and extensions.

Object-oriented software — if properly designed — can use the same principle, separating concerns into different classes.

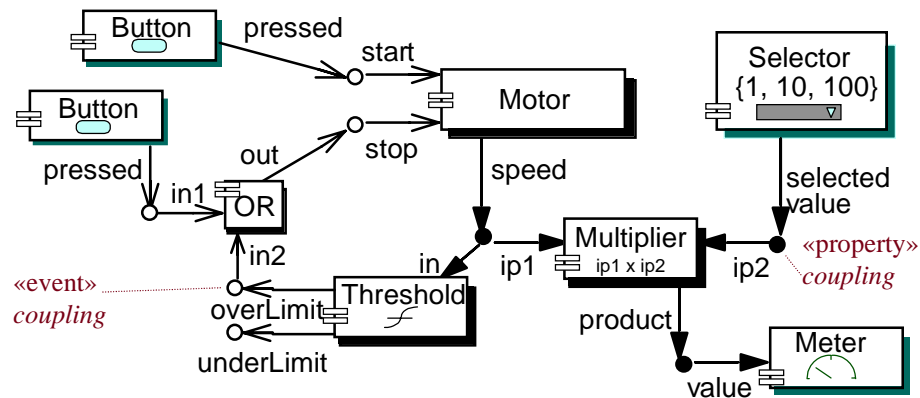
4 Where do we see components ?

At different scales and interconnected with different technology.



5 Components, kits, and connectors

Hardware designers have been making systems from pluggable components for many years: most of the pieces inside the average car or telephone have not been specifically designed for that one product. Let's look at an example that could be either a hardware or software component-based design, and see what lessons there are to learn from it.



This is a diagram of component-instances. The Buttons have outputs labelled **pressed**, which carry a signal each time the user presses the Button. The Motor might be either a real hardware motor with three labelled wires attached to it, or an equivalent piece of software controlling a motor; the **start** and **stop** wires are inputs accepting signals of the same type as the Buttons' outputs. The Motor's **speed** output carries a different kind of signal, a continuously varying numeric property. The Meter continuously displays to the user the current value at its input, also a continuous numeric property.

Each of these components can exist in isolation, unconnected to others; each has some labelled inputs and outputs, of different types. The components' classes might all have been designed by different people; a further designer has instantiated and plugged together this particular configuration.

Connectors The simple concept of an interface as a list of function-calls is insufficient to describe these couplings: such interfaces describe only invocations a component can deal with. Every Motor has an output labelled **speed**, that can be connected to any input accepting continuous numeric updates: the connectors depicted here include the idea of outputs as well as inputs. The connections include the idea of differently-labelled inputs and outputs being coupled (e.g. Motor's **speed** to Multiplier's **ip1**).

Connectors of this kind are much more powerful than function-list interfaces.

Kits Notice that there are fewer types of connection than components: the Motor could be wired straight to the Meter, or one of the Buttons straight to the Motor's **stop** input. This is what makes this bag of components a coherent kit: its members can be plugged together in many different ways, just like Lego. If you're familiar with the Unix standard library, you'll recognise the same characteristic. By contrast, a collection of components procured independently would be unlikely to form a kit in this sense, since there would be few workable configurations not requiring a lot of 'glue'.

6 Component Kit Architecture

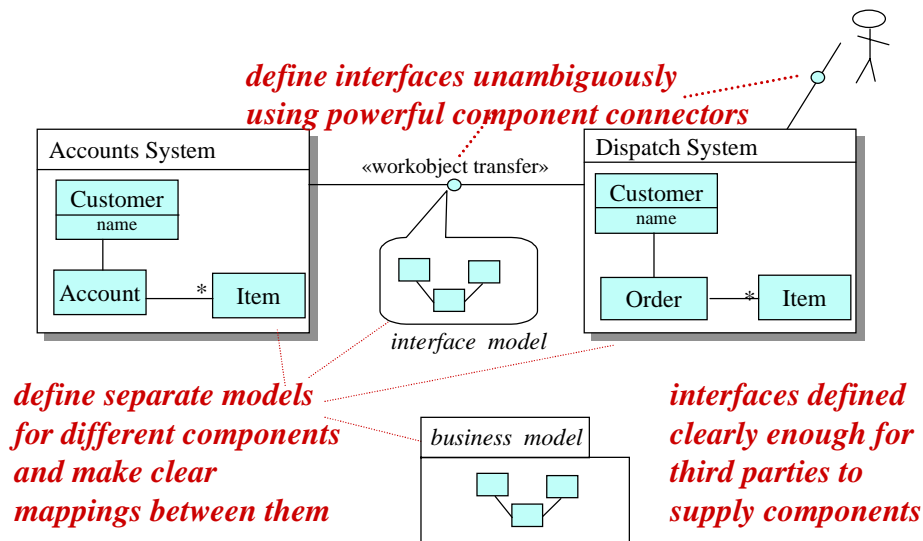
The same principles apply at any scale. In a larger system, the components will be substantial applications, possibly operating in different machines; and the connectors will be more sophisticated and carry more information. For example, a share-dealer support system produces a stream of deals; these flow through a variety of components.

In a component based design, we can separate three activities:

- Product building — rapid building of products by configuring components.
- Component design — implementing a specific component.
- Kit Architecture — defining the connectors.

Common connectors

The kit architecture has to be determined before any components can be completed. It defines how the components talk to one another: what kinds of connector there are, and how they work. In the Buttons and Motor etc example, the architecture defines what it means to have a continuous numeric output, for example: it might entail keeping a pointer to every coupled input, and sending a message whenever the value changes. There are several ways it could be done; the important thing is that any kit settles on one protocol for each chosen type of connector: this makes for flexible configurability. The number of connector types should be minimised, for the same reason.



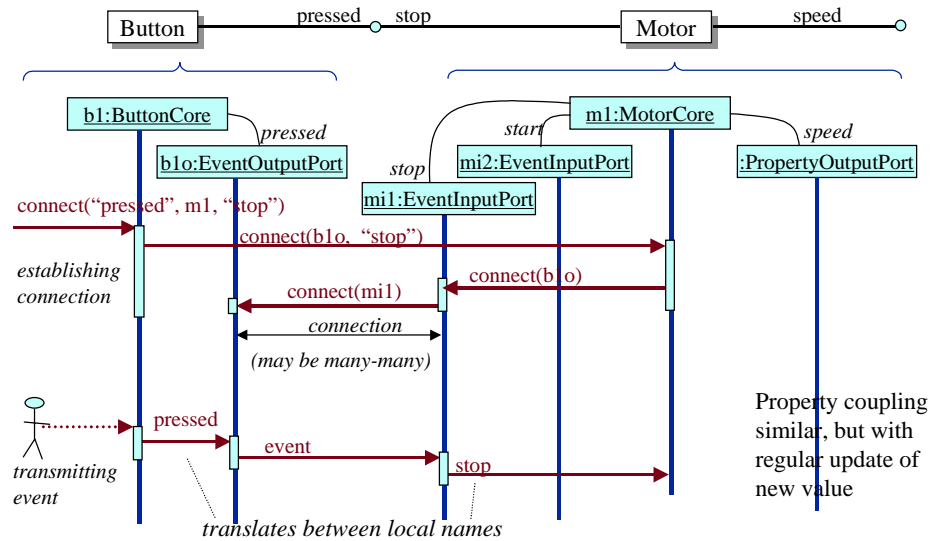
Common models

For larger designs, the information transferred between components is more substantial than just numbers. The architecture therefore includes a model of what flows across the connectors — what the components talk about. This is clearly related to a common business model.

Designers of components need to relate their own views back to the common model. For example, Accounts may think of a Customer as something with an account containing a list of debits and credits; Dispatching thinks of a Customer as a thing with an Order containing a list of products.

Defining connectors

Each connector is an abstraction that hides considerable implementation detail: it is above the level of individual object-oriented messages. Furthermore, there are many ways of achieving the same functionality — so this just shows one possible scheme.



In this example, we first see that each component turns out to be made up of several objects: in these examples, a ‘core’ object responsible for the functions of the component, and a separate object for each port.

First we see some third party instructing the Button to connect its “pressed” output to the Motor’s “stop” input. The Button knows the mapping between labels and actual port object, so it passes the message on to its “pressed” port object; which in turn applies to the Motor, which passes the request on to its own “stop” port. (We assume it’s the core objects’ business to know which objects are its ports; the rest of the world must just use their labels.) The two ports can then communicate with each other, and arrange to be connected. In particular, the output port keeps a register of all the input ports interested in anything it has to say.

Later, when the Button is pressed by a user, it gets its output port to send a message to all interested inputs; the message sent is one that is common to all ports of this type — which is what enables them to talk to one another, despite the difference in labelling between the connected ports.

This scheme achieves the componentware goal that reconfiguration is possible at run-time: that is, no recompilation or special programming is required to make the new connections. This means that each component can be a complete black box, with no source available.

There are of course other schemes that would have the same effect; but components can only be plugged into others that conform to the same scheme or “component architecture”: we say they belong to the same kit.

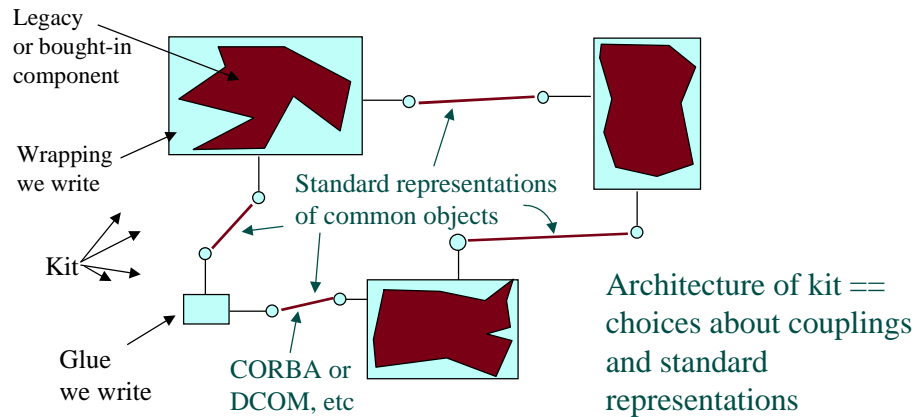
There are several types of connector. The «property» connectors are connected in the same way, but send messages every time some change in a given value occurs (such as the motor’s speed). There are of course, different types of value, so each port has to check at run-time that it is being plugged into a compatible port.

Kit architecture

Component kit architecture means defining a set of connectors that your projects will use, and defining what they achieve and how they work. Some of them can be adopted from existing platforms — for example, Java Beans supplies a mechanism similar to our continuous-update connectors; and CORBA provides a transaction service.

The point of the architecture is to define what conventions the component designers will stick to, in order to make the components easy to reconfigure.

Heterogeneous components



In the light of the discussion about kits of components, we can get a better understanding of the systems integration job we started with.

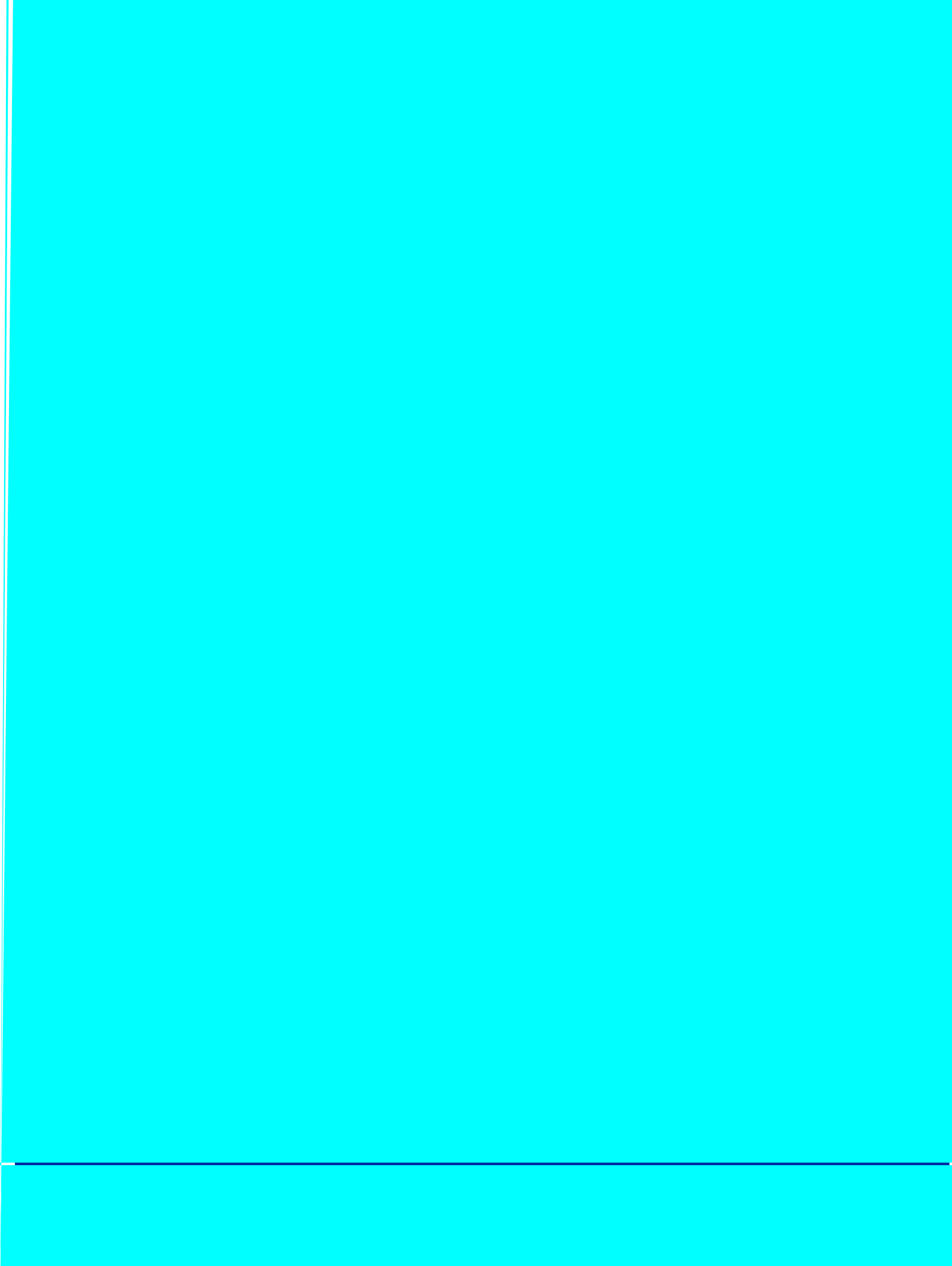
Schematically, the task is to "wrap" each of the diverse components so that it forms part of a kit, with couplings that will work together.

There's a short-term/long-term choice here: either we can wrap them just sufficiently to work in this particular configuration; or we can put in a bit more effort, and make the couplings general enough so that they will work in a few different ways, and perhaps with some other related components we know about.

A Kit --- even if it's only going to be used in one system --- always has an Architecture: a set of choices about how the interconnections will work, and how the different internal representations of what (say) a Customer is, are converted into a mutually understandable representation for transmission through the couplings.

A well-thought-out, clearly-defined architecture emerges as the most essential prerequisite when considering using components of any kind.

(Further detail, and a substantial worked example, can be found in:
<http://www.trireme.com/catalysis/book/>)



8 Wiring

Components can be connected by a variety of mechanisms.

Function calls. Objects within a single execution space communicate by means of function (= procedure or subroutine) calls. In OO languages, the same function name can have different implementations in different objects.

RMI (Remote Method Invocation). A mechanism for sending messages between objects executing in different internet hosts. Originally a general term, now hijacked by Java's implementation; both ends have to be written in Java, and they have to know where to find each other.

COM (and DCOM, COM+, ...). Microsoft's standard for communication between Windows™ applications. The programs can be written in different languages, and can be in different hosts.

CORBA. The OMG standard for communication between object-oriented components. Like COM, but not restricted to Windows.

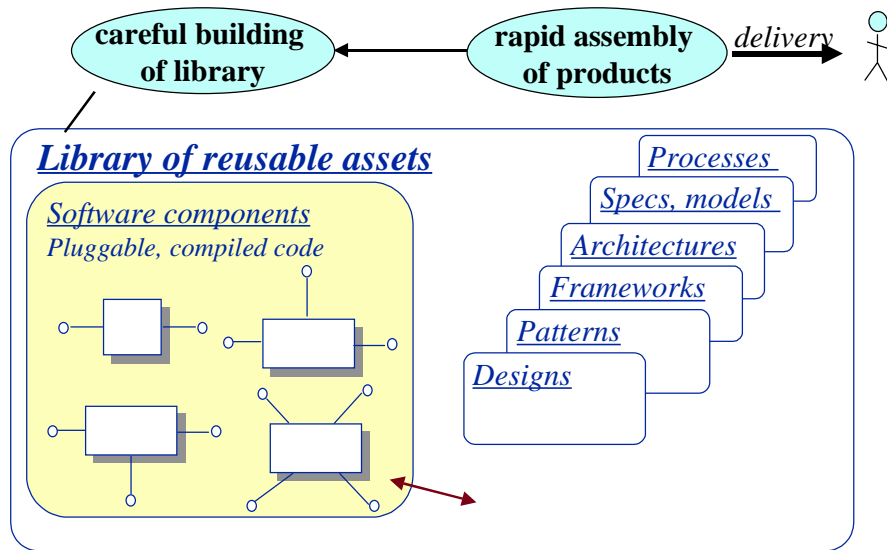
CORBA and COM differ in many details. CORBA is more coherently defined, but fewer implementations can be found.

Older wiring

There are many other ways in which components may be connected. They do not generally come with a defined mechanism for defining or checking interfaces.

- Inter-process events or signals — Apple, Windows, Unix all provide them
- Pipes — Unix stream communication between processes
- TCP/IP — inter-host protocols
- File transfers; email — for example built on TCP/IP
- Shared access to a database — components interact via database records

9 Reusable assets



Many of the artifacts of the design process can be generalised and used in more than one context.

By 'reuse', we don't mean cutting something out of an old design, adapting it and pasting it into a new one. We mean making something that can be used as is in several different contexts. This shares the cost of future improvements. Because the generic asset is to be used in many contexts, it is worth investing much in its quality and functionality. This benefits all the end products in which it is used.

Reusable assets include patterns (pieces of advice on how to solve design problems); specifications and parts of them; even ideas about the design process.

We reserve the term 'component' for those assets that the client designer does not have the opportunity to change: most effectively, executable code without the source. This makes for much better version control and more effective subsequent maintenance. It also means that the originating designer can sell a solution without giving away how it works.

Reuse management

A software house that plans for reuse separates different activities:

- Architecture — planning a coherent kit of reusable assets that will fit together well and can be used to create many software products with a business domain. This is an ongoing activity that enhances and extends the software capital of the organisation.
- Asset-building — implementing a specific component or other asset that fits within the kit architecture. Includes generalisation of designs originally conceived for specific end products; and wrapping legacy and bought-in assets to fit the kit architecture.
- Product-building — rapid building of products by configuring assets from the library, together with the design of specialised components not in the library.

Minimise dependencies

Having separated concerns, we should keep them as independent as possible, with as few restrictions as possible on the allowable combinations. Try not to build a Bonus Pay component that depends on the installation of the Weekly Laundry component. That way, we minimise the number of rules we need to deal with, about how to configure the system.

On the other hand, do build meaningful dependencies into your components: the Bonus Pay component presumably depends on there being a basic payment component in place; but let it work with any payment component, not just a specific one.

In short, a component should be dependent on just those others that it needs to be, and no more.

Modifications not allowed by configurers

The variability should be, as far as possible, contained within the configuration of components. Strict control should be imposed on changes to the components themselves, which should be performed only by the originators. Divergence of different versions should be prevented. Backward-compatible extensions in functionality should be preferred to changes.

When one set of users require a change, it is often possible for them to ‘wrap’ the supplied component within an extension of their own. The new functionality, once tested at a local level, can be incorporated into the official kit. (This often happens with operating systems: someone invents a neat idea and runs it as a local add-in, and then it becomes adopted in a later edition of the OS.)

Builder tools and 4GLs

Once a connection technology has been established, it is possible to make tools for building a system from components. There are many visual builder tools for GUIs: you draw the screens and the tool hooks up the code (though often in a very flat fashion).

There are visual tools for building software with Java Beans: you draw the program and the tool plugs the components together. VisualAge does the same job with a different kit of components, and with the extra benefit that the same visual program can be implemented in Java, C++ or Smalltalk at the touch of a button. Many of the workflow systems also provide a pictorial language for programming the flow of objects between work-processing components.

At the distributed level, there are also builder tools such as Forté and Dynasty. Many of them can work with a mixture of interconnection technologies.

10 Components compared with ...

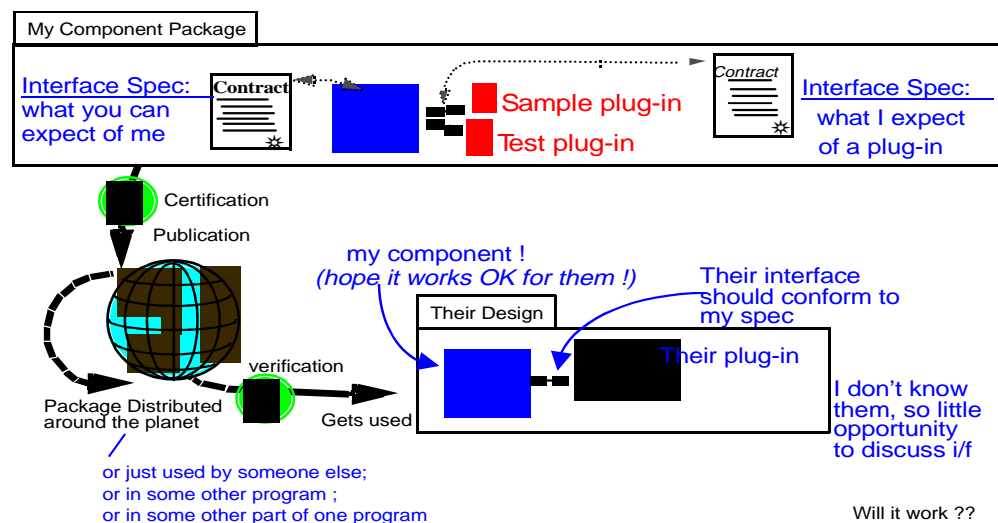
Components are more than modules

Software engineers have always divided large pieces of work into modules that different designers could work on. Separation of concerns meant that each designer did not have to know everything about how other modules worked. However, the separation was not strict, and dependency of one module on the internal design of another was common. Furthermore, if there was any question about the interface between one module and another, the designers could easily talk to each other.

Components are polymorphic modules: that is, each component is designed to be able to work with any other component that conforms to a defined interface (just as, for example, your computer works with any monitor screen conforming to the defined standards). This enables many component-designers in different parts of a large organisation, or indeed the world, to contribute to a kit of components; and for other designers to assemble the components into products; and for none of these designers to know each other.

When you design a component, you do not know what other components it is going to be plugged into. You may have some examples to hand, but some of your clients will plug your component into another that you have never seen, and perhaps which does not exist yet.

This means that the big difference between modular and component based design is that we have to be very careful to define interfaces unambiguously. It's extra work, sure; but we will get a return on the investment once components start getting used more than 3 or 4 times.



Components aren't just objects or classes

Meaning of object, component, instance, class

First of all, let's get some vocabulary agreed:

- Object instances are created from object classes. An instance runs in a particular machine over a particular period of time; a class is a piece of program. We sometimes loosely talk about objects when we might mean either instances or classes.
- Component instances are created from component classes. An instance runs in a particular machine over a particular period of time; a class is a piece of program. We sometimes loosely talk about components when we might mean either instances or classes.

Common between components and objects

- Encapsulation. Both encapsulate a related set of services and the data necessary to provide those services, and hide the implementation details behind a well-defined interface.
- Inheritance. A class can be defined as an extension of the code of a common implementation. Similarly, a component class can import code defined in another package.
- Interfaces. Both objects and components should have well-defined interfaces. Many classes (or either kind) can implement one interface, and several interfaces can be implemented by one class. Because of encapsulation, one object or component may work with any other that implements the appropriate interface.

Distinctions between components and objects

All of these are fuzzy:

Scale.

A component is usually compiled separately, has its own process. If an object contains a reference to another object, it will typically be in the form of a memory pointer; if a component refers to another, it may be an IP address, which needs some underlying apparatus to interpret it. If an object sends a message to another, it will typically be done with a function call with a few primitive or pointer parameters; if a component interacts with another, it may be a complex transaction and involve the transfer of large volumes of information. Each service may be a complete separately compiled program. The information transferred in an inter-component communication is not generally represented as a reference to another component.

Persistence.

An object is usually a set of functions in an executing program, with a state represented in part of the machine's main memory. If the machine is switched off, the object goes away. Special programming is needed to make an object migrate between memory and disk, and it needs to be in main memory to be executed.

A component may be a set of application programs whose common state is a database.

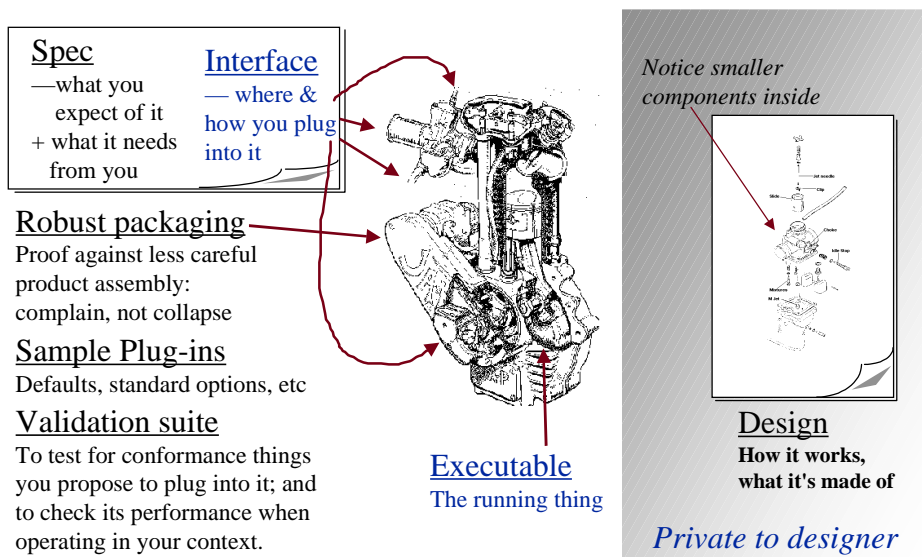
Robustness

An object's designer will typically document a precondition for a function, and expect the client to ensure that it is met: if not, the subsequent failure is their fault; it is too costly to program defensively against all possible problems.

A component has much weaker preconditions – ideally, none at all: it can do something sensible with anything that is thrown at it. And anything not meeting the precondition is dealt with predictably: no falling over.

This difference is because the activity of assembling components into products is much more rapid and done with much less checking than the construction of components. In product assembly, the idea is to put together, in as short a time as possible, a relatively short-lived configuration that will satisfy the end users. Product assemblers may not bother with requirements specs, and may configure components experimentally. Hence you do not know what other components you will be talking to.

Packaging A component should come with a test harness, documentation. It might not come with source code.



Complexity of interfaces An object's interface is usually characterised as a list of function calls. A component has much more complex interfaces to the outside world, so that the function call level is really insufficiently powerful to describe it. We will shortly introduce the idea of component connectors, which include notions of outputs and complex transactions.

Distribution Designing object collaborations in a distributed system has two main differences from objects within a single execution space. Firstly, a component may go offline unexpectedly, and the others will be expected to keep working. Secondly, traffic between components must be economised. (Other problems of communication, such as security, accuracy, and routing can be confined to the lower interconnection layers.)

Other Mechanisms for Separating Concerns

Parts of the behavior that should be or may be variable, should be in different parts of the design, that can be altered separately. What is the nature of these 'plug-in parts'? There are many solutions. Object-oriented programmers immediately think of objects, but it is worth recognising that there are other approaches that are valid in many cases, particularly in a context where there is much legacy software around (or perhaps just a legacy skills base).

Parameters For variants such as how many rooms there are in the hotel, we need only an initialisation procedure to enter a few numbers. The plug-in part is just a number, and the socket is the field where you enter it.

Tables On a larger scale, we can make the software table-driven: the plug-in part is the table. A drawback of tables and parameters is that the principal component has to include all the variant behaviour that is anticipated: the tables are just data, and the software typically contains lots of case-statements.

Interpreted language For more complex variations, we can invent a specialised language — for example, a Hotel Definition Language in which we can define the procedures for admitting guests, and the rules for paying staff and doing accounts. (It could be a conventional text language, or a pictorial one such as many 4GLs provide.) The plug-in parts are the programs written in this language, and the principal component works as an interpreter of it.

Defining such a language is a skilled business, often underestimated. The facilities demanded of it nearly always grow well beyond the original vision: at which point, the language becomes a mess, opaque to newcomers and fervently supported by those familiar with it. (Look at Perl, for example. Or many of the proprietary workflow languages. Or if you're familiar with C++, just stand back for a moment and try to justify the existence of both pointers and references.)

In general, it is better to choose an existing language from the start (so that Hotel Definition Language is a dialect of Visual Basic, Ada, or Smalltalk); and not to embark on this path except for a truly grand project.

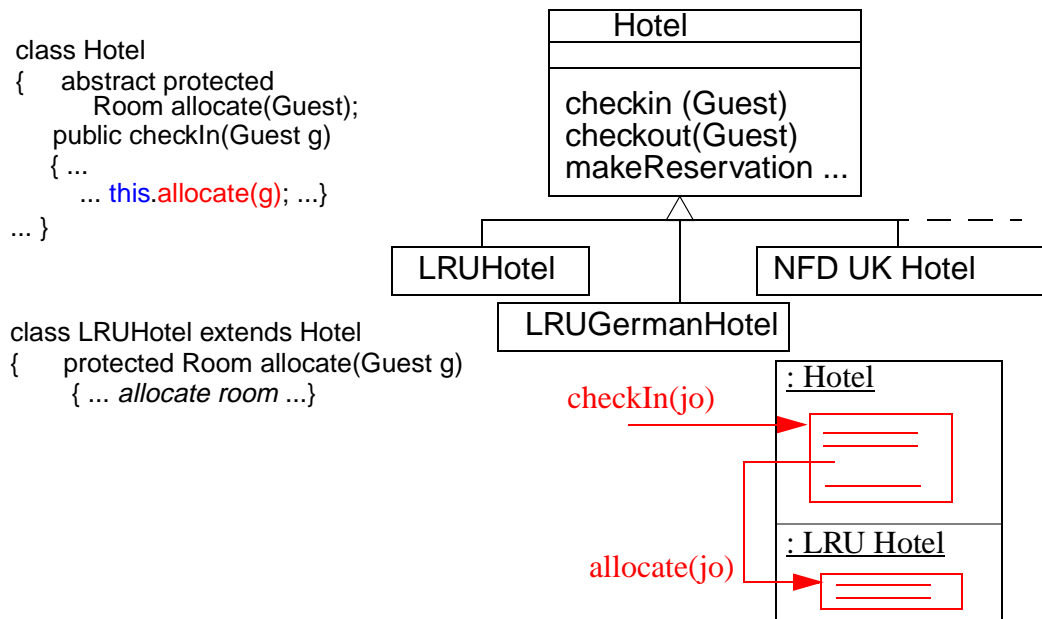
Pluggable chunks of program code This is the solution we shall focus on. A completed system is built from modules that fit together, in the same way that hardware is built. To build a system, select components from the kit, and plug them together.

Object-oriented inheritance

OO inheritance is a particular case of a not-very-good method of getting flexibility.

Within the Hotel support software, we could represent a Hotel with an object; different variations of the basic behaviour could be made by defining a separate subclass for each Hotels that allocate the least recently used room first, for hotels that pay their staff according to German legislation, etc. The program code in the Hotel superclass does most of the work, but passes control to the subclass whenever it needs to allocate a room, work out pay, etc. Thinking of each object as taking part of its definition from

its class and each superclass, the behaviour is shared between the more general half of the object and the more specialised.

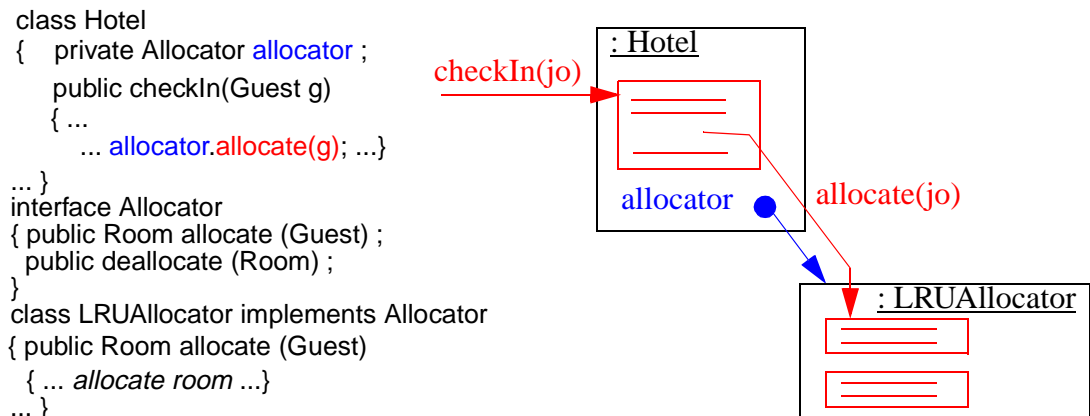


But this strategy is rather limited: there are other variations (such as methods of staff payment), and you would have to write a separate subclass for every required combination; which in turn makes it difficult to create ad hoc combinations as new objects are instantiated. Furthermore, there is no possibility of changing during an object's life (to allocate rooms differently at weekends, for example).

Object-oriented pluggability

We can illustrate pluggability within the context of an object-oriented program.

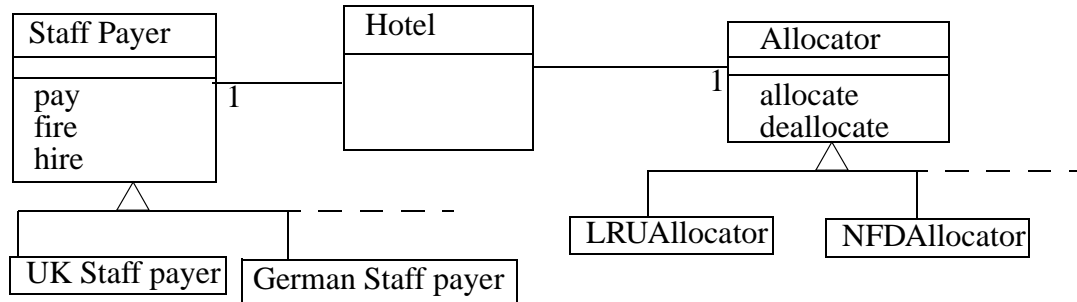
On instantiating a new Hotel object, we also create and plug in a room allocator, staff-payer, and so on. For each plug-point, there is a selection of classes to choose from. Connected objects keep pointers to each other: 'plugging in' just means updating the pointers.



The behaviour of the Hotel can easily be changed, by instantiating a different class of Allocator, and pointing to it from the Hotel.

Each plug-point requires the definition of an interface — a list of the messages that can be sent between the connected objects. In Java, the language has an interface con-

struct just for this purpose. (C++ pure abstract classes are used the same way.) You can declare a variable or parameter to hold objects conforming to a given interface (Allocator in the example), thus defining part of the behaviour to be expected of any object found within that variable. You can then assign to it an instance of any class that implements that interface.



So the tactic is: whenever you see a possible variation in behaviour, move all the relevant functions and data out to a separate object or ‘role’; and define an interface to it.

(In OO design, there is a more general strategy, that the classes you design with should be based on business concepts such as Room, Guest, Staff_member, etc. This takes care of the fact that we can’t always anticipate every possible future behavioural variation. The business concepts act as a default separation of concerns, onto which the roles can be mapped: for example, the functions of Staff_payer could be assigned to Staff_member.)

The same principle of pluggability (or ‘polymorphism’) applies not just to objects in an OO program, but to large components, where each component may be a sizable program in its own right.

Large scale A key to successful connection with any of these mechanisms is the definition of the interface. This is at minimum a list of the messages that can be dealt with by a component: it represents part of the contract between user and provider of a service.

CORBA and COM each come with an Interface Definition Language (IDL); Java has an language construct for the definition of interfaces; and in C++, the pure abstract class serves the same purpose.

Once an interface has been defined, you can start defining components that use it; while other designers can write components that implement it. The compilers will check that you aren’t using functions not advertised in the IDL declarations; and that the implementors are providing functions for all the advertised declarations. The compilers cannot, of course, check that the functions are being used properly, nor that they provide the correct behaviour: this is up to the designers themselves to document and check carefully.

(Smalltalk is an example of a language that does not have interface definitions, and does not have compiler type-checking. This just means that there is slightly more work for the designers to do.)

11 Summary

- The basic principle of component based development is polymorphism, or pluggability: that variant behaviour is produced by reconfiguring & parameterising components which themselves have fixed designs. The same principle applies on any scale. There is a variety of technologies by which the components can be composed.
- Components should preferably be designed in kits. A kit is a coherent set of components, with a minimal set of interfaces.
- There are three principal activities in component based development, requiring different balances of maintainability and cost:
 - Kit architecture: defining common models and interfaces
 - Component development
 - Product assembly from components
- Interfaces defined as lists of function calls are too low-level for component-based design. It is better to design with more abstract connectors, which include different kinds of transfer or transaction. The choice of connectors used in a particular kit are part of its architecture; the ‘wiring’ may support some of them directly.
- Each component may have its own partial view of the complete business model. The design method must include ways of mapping these views.
- Components may be distributed across different machines. In this case, the design method must include patterns that deal with the possibility of links and machines going out of service, and treat links as potential bottlenecks.

In order to achieve good component based development:

- The structure of the software producing organisation must be such as to resource the three development levels (architecture, components, products).
- The teams must be appropriately skilled in an appropriate design method, that includes appropriate patterns for component and distributed development; including the notation and means to define connectors, to define components, and to build products from components.
- Select appropriate tools to support the design method; and select tools and platforms to support the architecture.

12 *Further reading*

Clemens Szyperski, *Component Software*, Addison-Wesley 1998 0-201-17888-5
— Highly recommended.

<http://www.omg.org> —pOMG web site, for CORBA details.

<http://www.microsoft.com> — Microsoft web site, for COM details.

<http://www.tireme.com/catalysis> —pCatalysis CBD method

Desmond D'Souza and Alan Cameron Wills, *Catalysis: Objects, Frameworks, and Components in UML*, Addison-Wesley 1998 — more on the topic of this paper