

Notes from Well House Consultants

These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit

*<http://www.wellho.net/net/whcotnl.html>
for details.*

1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

graham@wellho.net

technical contact

lisa@wellho.net

administration contact

Our full postal address is

404 The Spa

Melksham

Wiltshire

UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

Object Orientation: Design Techniques

This is the third and final module that introduces object orientation, and is applicable to any language with OO support. It is vital that applications are correctly designed from the beginning. Starting from first principles, we take students through the steps of modelling a system using informal methods, and we also introduce more formal methods such as UML.

<i>OO Design – some basics</i>	4
<i>Informal Techniques</i>	5
<i>Unified Modelling Language (UML)</i>	13
<i>Tools</i>	19
<i>Project management and design issues</i>	19
<i>Extreme programming</i>	20

Before you start to write code ... or classes ... or methods ... you need to give some thought to what you're going to write, every bit as much as you need to plan a holiday away from home before you actually set off on your travels.

The more your travels vary from the norm, the more planning you have to do ahead of time. Even if you're away for just a few days, some trips can be weeks or months in the planning; once the planning's done, the actual going is the easy bit!

Any new object oriented project needs a great deal of careful planning and review before a single line of code is written. The end user requirements need to be learned and considered, and indeed it's up to the analyst(s) doing the planning to have more foresight than the end user – to ask those searching questions, to think about what the future may hold, etc.

As you might imagine, there are a number of informal and formal techniques for planning and maintaining object oriented projects, and a number of tools of various sorts that are available to help you implement the techniques. Some are as simple as pencil and paper, others are complex and expensive pieces of software.

2.1 OO Design – some basics

Start with a good understanding of OO

If you're taking this module because you'll be working on a major OO project, remember that you do need to have a thorough understanding of Object Orientation, to know your encapsulation from your polymorphism from your privates, before you start. Otherwise you'll probably end up throwing out your first attempt!

Find a small training project. Design it, implement it. Learn OO that way first before you jump in with both feet. Then get a good understanding of the project.

One of the many OO authorities [Coad] came up with three development cycles:

Waterfall –

- Analysis
- Design
- Programming

Spiral –¹

- Analysis, prototyping, risk management
- Design, prototyping, risk management
- Programming, prototyping, risk management

Incremental –

- A little analysis
- A little design
- A little programming
- Repeat

Although it may seem counter to what you're used to, and managers may worry at the lack of coding until late in the project, the author of this course has found the waterfall cycle very effective in some circumstances, with even the user manual written before a single line of new code exists!

And also a good understanding of the buzz words and design cycle.

From a web site Useful text, once you're a little way there! ...

OOA and OOD stand for Object-Oriented Analysis and Object-Oriented Design, respectively. OOA strives to understand and model, in terms of object-oriented concepts (objects and classes), a particular problem within a problem domain (from its requirements, domain and environment) from a user-oriented or domain expert's perspective and with an emphasis on modeling the real-world (the system and its context/(user-)environment). The product, or resultant model, of OOA specifies a

¹ The spiral model is often incremental and may waterfall if called for

complete system and a complete set of requirements and external interface of the system to be built, often obtained from a domain model (e.g. FUSION, Jacobson), scenarios (Rumbaugh), or use-cases (Jacobson).

A new Unified Modeling Language (previously Unified Method) is now being worked on by Grady Booch, James Rumbaugh, and Ivar Jacobson at Rational Software which should be made into a public standard, perhaps to be adopted by the OMG. The latest docs can be found online from the Rational home page.

The usual progression is from OOA to OOD to OOP (implementation) and this Universal Process Model roughly corresponds to the Waterfall Model [Royce 70]. See [Humphrey 89] and [Yourdon 92] for a few of many discussions on software life-cycle models and their use. Humphrey also details Worldly and Atomic Process Models for finer grained analysis and design in the Defined Process (see below) and discusses other alternatives to the task oriented models.

2.2 Informal Techniques

This section talks you through some of the things you need to think of as you design your Object Oriented system. We'll then go on to have a look at some of the more formalised techniques, concentrating on UML.

Micro or Macro?

You have to start somewhere. Do you start your design with the detail of the smallest data objects that you'll be using, or with the overall system and the classes that you'll be using in your main application?

Since there's an interface between each of the classes that you'll be using, some degree of flexibility is called for. Starting at the Micro level, you need to specify your user interface by analysing the next level as you specify the current level! Starting at the Micro level does mean that you can write test classes at each level as you write them. A series of thorough test harnesses, level by level, should ensure that problems are located in the detail before they can fester and spread.

A disadvantage of the Micro approach, though, is that you need to keep your eye on the overall target. It's very easy to specify and write classes, build them into other classes, but then discover that the classes you've written can't actually be bolted together to do the job that your end user needs.

Specifying classes and methods

What do you need to consider?

- Variables per instance
- Variables per class
- Method calling specifications

At the very least, write out lists and draw a diagram such as shown in Figure 1 and this will form a basis for you to then go on to some of the more formal methods later.

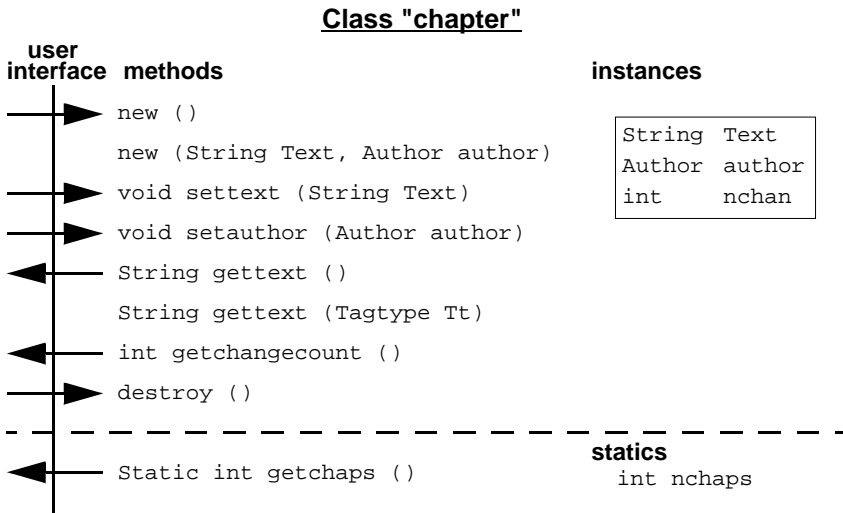


Figure 1 Within this diagram, we show: on the left, the user (caller's) interface to the class; on the right, the variables associated with each instance; at the top, instance methods and variables; at the bottom, static variables and methods.

Having drawn the basic diagram, the way is open for me to produce other diagrams; I can add lines showing which methods use which variables, for example as shown in Figure 2.

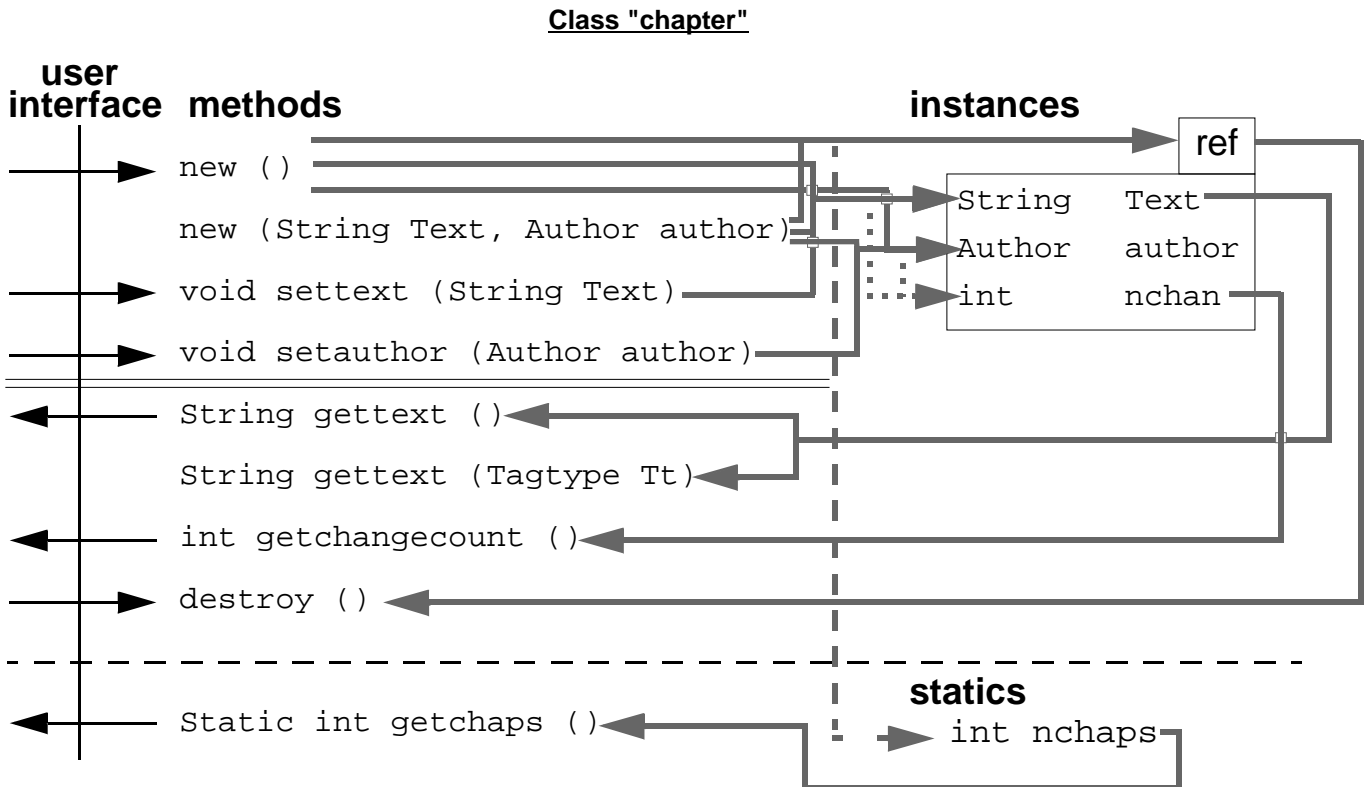


Figure 2 Solid lines on this diagram show which methods are directly reading or writing which particular instance variables, and dotted lines show how we're proposing to access other variables that are held within each instance or within the class.

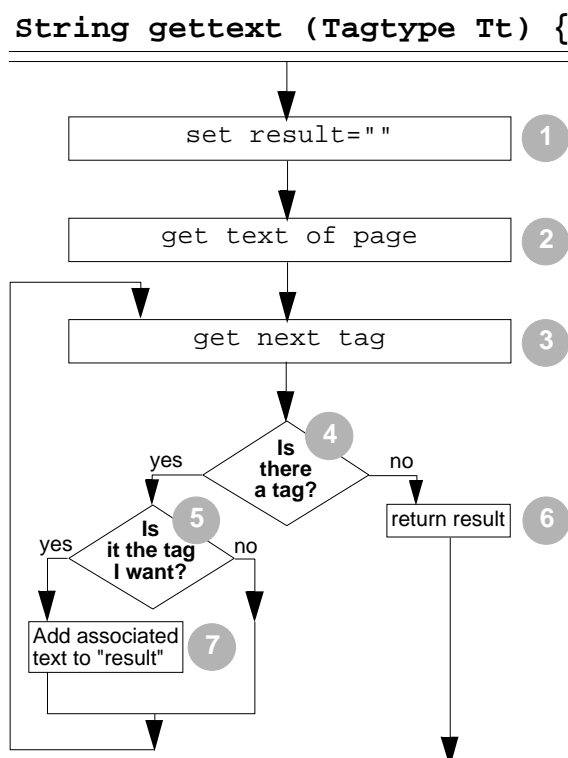
We're designing from first principles here, and if I follow this approach with a more complex class, showing extended information on the chart, my diagram will become overcomplex. That's when I need to start using a tool that lets me highlight certain features, or show certain aspects of the (as yet!) informal model I've started to build.

– What else can I see from the diagram already?

I can very clearly pick out certain flaws in my class design. In the example shown, my user interface allows me to set the author, but no methods have been provided to access that data!

If I want to see how a particular variable is used within the class, I can see all the influencing methods.

Figure 3 A flowchart of one of the `gettext` methods



In this particular example, I can see how the design neatly divides the methods into those which set variables and those which retrieve information – the **set** and **get** approach that's used by Java Beans. I've added a double-ruled line to my diagram to show you this division.

What can't I see from the diagrams I have?

– How the individual methods operate

I can't see how the individual methods work nor what algorithms are involved. For such work, I might use a traditional flowchart, I might write pseudocode, or I might even start writing real code in my target language. By using or calling a number of private methods within the class, such target code should be virtually self-commenting.

In the pseudocode examples that follow, the comment numbers apply to the box on the flowchart.

Pseudocode - near-Java

```

public String gettext (Tagtype Tt) {
    StringBuffer result = new StringBuffer(""); // 1
    StringBuffer pagetext = new StringBuffer(Text); // 2
    Taggedtext ttex;

        while ((ttex = pagetext.gettagged()) != null) { // 3,4
            if (ttex.gettype().equals(Tt)) { // 5
                result.append(ttex.getchars()); // 7
            }
        }
    String rr = toString(result);
    return (rr); // 6
}

```

Pseudocode - near-Perl

```

sub gettext {

    my ($inst,$tt) = @_;
    my $result; # 1
    my $ttex;
    my @ttex_list = gettagged($inst -> "text#)2
    foreach $ttex(@ttex_list) { # 3, 4
        my ($ttthis,$charsinthis) = splittagsection($ttex);
        $result .= $charsinthis if ($tt eq $ttthi#)5,7
    }
    $result; # 6
}

```

My earlier diagram could be extended to show private methods too. They would be placed to the right of the public methods and arranged in a hierarchy similar to the calling hierarchy.

– The State of an object

The diagrams that I've drawn so far are both views on my perception - my model - of the class that I'm writing. But there are also other aspects to consider that might be implicit prom or partly answered by what we've written so far, but are not obvious from either view or diagram.

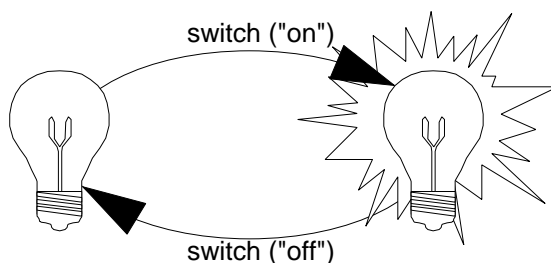


Figure 4 A state diagram to show a light bulb on and off

Let's consider one more feature of an object – an object can be in a particular condition, or have a particular state (or a series of states). A light bulb can be lighted or not lighted. I can draw a state diagram to show each of these states, and then add lines to my diagram to show how it moves from one state to another.

It's now easy to see how you can move from one state to another, and also to

spot any states that you can get stuck in. Indeed, a state diagram is an excellent tool for designing web sites and for using to follow visitors through the site from your logfiles.

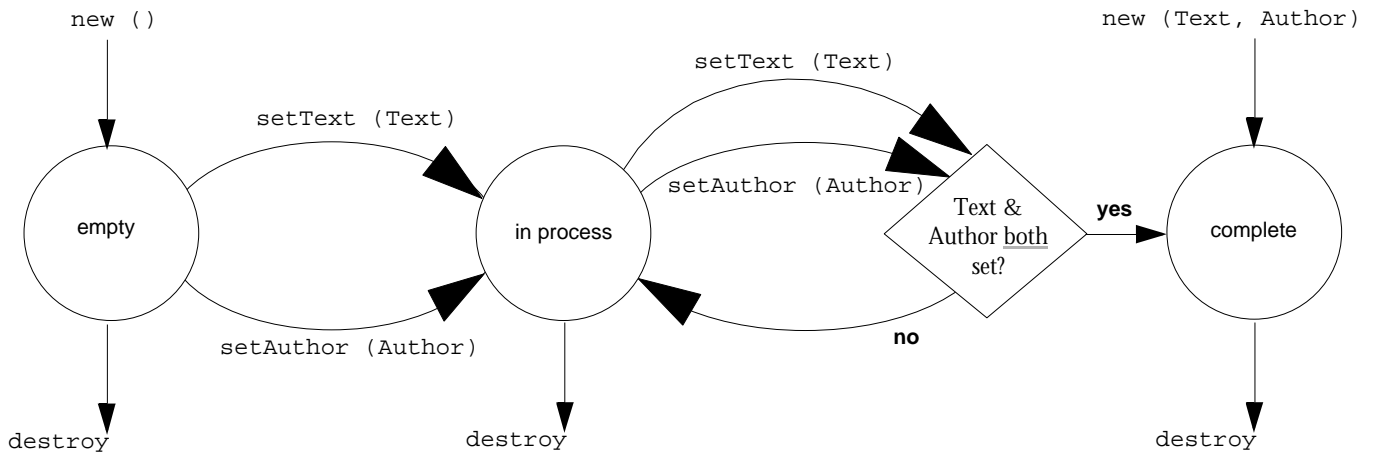


Figure 5 States of chapter object

Even with our "chapter" class, where we either have a chapter or we don't, a state diagram can be drawn (see Figure 5).

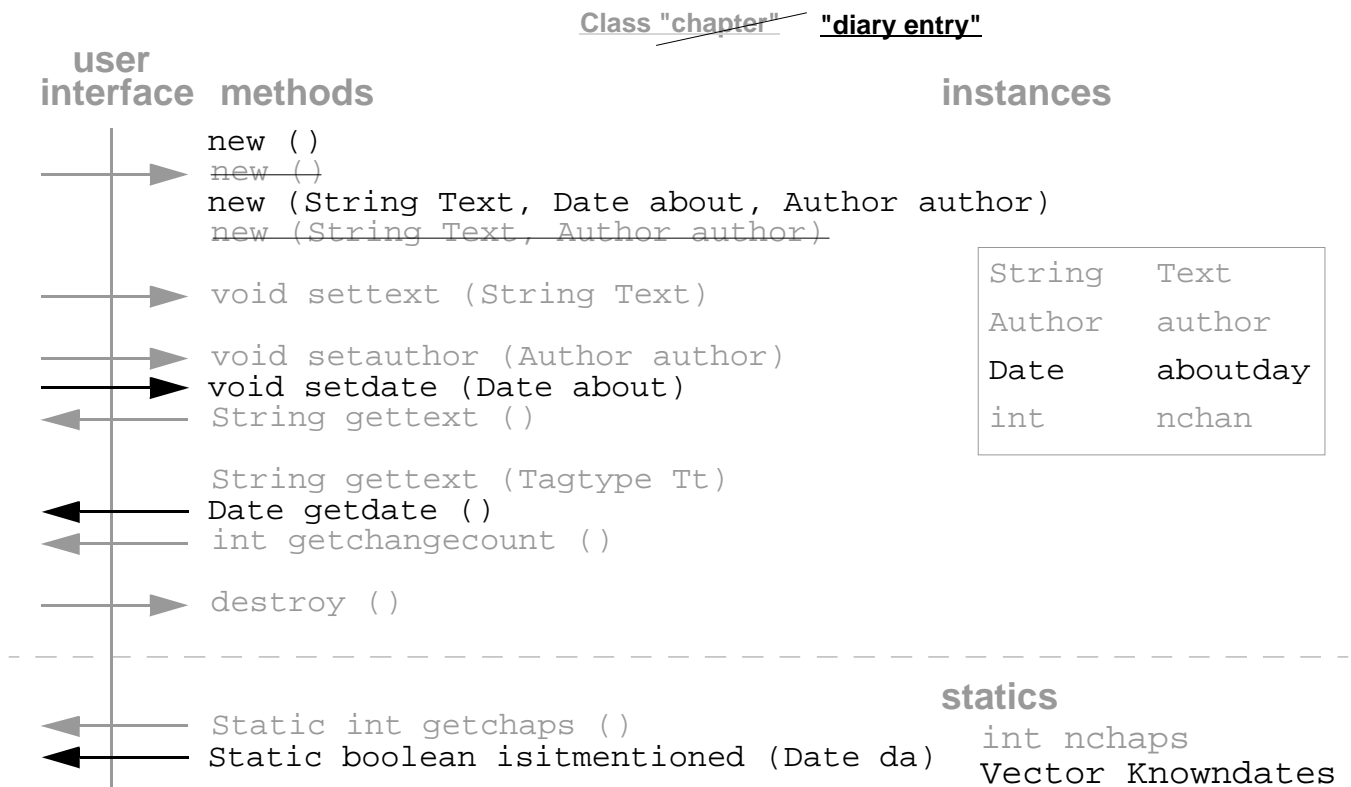
Specifying extended classes

What goes into the base class, and what's extended, and what's abstracted?

Your model needs to reflect the real life situation ...

Once again, you can extend the diagrams that you drew earlier for extended classes. Let's take our objects of type chapter and extend them into the diaryentry class.

The following diagram shows the original class in grey and the extended information in black.



We've added the extra instance variables necessary to the diagram, and additional methods. Constructors are not inherited, so we've crossed them out (in reality, we wouldn't show the constructors of the base class at all on this diagram).

Methods of the base class which are overridden should be shown with the original method crossed through on the diagram and the new replacing it.

So why did the date object go into the extended class rather than the base class? Because it's something that relates to a diary entry but not to other types of chapter. "What date applies to this chapter?" (of this course) is a nonsense question. There might be other date members, such as date written, date first presented, date most recently presented, date next revision due, etc ...

If you find yourself adding (nearly) identical class members (methods and variables) to a whole series of extended classes, then you should consider moving those members into the base class.

If you find yourself writing a series of (base) classes which have nearly identical members, then you should consider creating a single base class and extending it if and when necessary to create each of what started off as independent classes.

If you find yourself overriding a method in every extended class of a base class, but needing to vary how you do the overriding, then you should be looking at abstract classes [Java].

You're not going to get your design right the first time you put pen to paper. You'll be changing designs, looking at different views, etc. Good reason to be using tools of the type we'll describe later, and more formal modelling languages so that you can work alongside others on projects and the whole team can have a clear understanding of what other team members are doing purely by looking at their models.

Figure 6 Specifying extended cases

Clusters of classes

It's very rare for a single class to do everything you require. Within one type of object, you'll be using objects of another type.

If I'm writing a class of "chapter" objects (that could be extended to coursemodule, bookchapter, diaryentry, etc), I'll also need and call author objects, date objects (for when written, when published, etc), which will logically be separate objects rather than a part of the actual class. Indeed, the actual text of my "chapter" will be (in my example) a String object from the standard library of the OO language that I'm using, and I've used "Author" and "Tag" objects too!

The diagrams earlier in this section actually included the Author and Tag objects; not obvious to see from our black and white paper diagram, but using computerised tools, or using colour, such additional classes in the same cluster or elsewhere can be noticed, appropriate diagrams can be linked to, if need be all the variables within (say) the *Author* object class could be added to the diagram for a chapter, etc.

Generalise it out

"The best OO projects get simpler rather than more complex as they proceed". Amazingly, this has proved true many times over. And the more generalised your code is, the more it can be reused. Whoever wants to write two similar classes on two similar projects?

Programming and method standards

Before we look at some of the more formal methods which are around, a short comment on programming and design standards.

Perl especially, but also Java to some extent, allow you a flexibility over and above what you'll need within your particular organisation. You'll probably want to adopt particular programming styles and standards throughout your work or your team's work.

Programming standards as a whole are a quite separate topic beyond this module, but you should be giving thought to aspects such as ...

– Construct and populate separate?

As a matter of course, are we going to write constructors to set up an empty object, which we'll then initialise through a separate call, or are we going to write the two combined?

In our chapter example, we could elect to have all the work done by constructors:

```
thischap = new chapter(String Text, Author author);
OR thischap = new chapter(File loaditfrom);
```

OR we could construct an empty chapter:

```
thischap = new chapter();
which we then initilased or populated as follows:
thischap.init(String Text, Author author);
OR thischap.fromfile(File name);
```

Which is best? We could argue about this. Some authorities will tell you that the creation and initialisation are different jobs and should be done in separate calls. They'll point out that doing both in a single function means duplicated code if the object can be initialised in several ways.

Others will suggest you should always initialise objects as you create them, to avoid you accidentally calling inappropriate access methods on objects which haven't been initialised yet. In other words, if you combine the create and initialise, you save yourself a "state" that you have to consider in each of your called methods.

– **get** and **set**?

When accessing object methods, it's often sensible to separate out those which

store information into the objects, from those which analyse the information in the objects, from those which retrieve information. Usually the last two groups are combined so that you have:

- methods that store information and
- methods that retrieve information.

It's sensible to name to a convention.

The human reader who knows the convention can instantly tell which group a method belongs to.

Tools that read existing programs and produce models from them can have that same knowledge.

Within some languages (and this applies to both Java and Perl) there's a facility called introspection which lets the language find out the names of all the methods.

The "de-facto" standard is to use words like:

set to start method names that store information
get to start method names that retrieve values
isit to start method names that pass back boolean states.

and tools exist that make use of this convention; we're talking "beans" in Java.

Let's say that we're setting the author of a chapter. What set methods could we use?

Java

```
setAuthor(authorobject);
```

OR `set("Author",authorobject);`

and you'll find authors that use either (or both) of these standard conventions.

If you have a class that includes a large number of variables for each instance, many of which are just stored and retrieved, the second approach has some merit; the code author is saved the need to write or generate a very large number of methods, and indeed the single set method can be used to set several values at the same time:

```
set("Author",authorobject,"Text",contentstring);            Java  
set("Author"=>$authorobject,"Text"=>$contentstring);       Perl
```

On the other hand, a series of explicit methods imposes a more controlled user interface onto the authors of the classes calling your methods. The more explicit that interface, the more robust the code is likely to be and the less change there is of the user managing to call something in a way not intended by the author.

Formal Methods

Objects need to be designed, and somehow those designs written down. It's often done in a visual language, which should be

- Accurate
- Consistent
- Easy to communicate to others
- Easy to change
- Understandable

We've come across the following:

Berard
BON
Booch
Coad/Yourdon
Colbert
de Champeaux
Embley
EVB
FUSION
HOOD
IBM
Jacobson

Martin/Odell
 OOram
 OMT
 OOSE/Objectory
 ROOM
 Rumbaugh
 Shlaer and Mellor
 OPEN
 UML
 Wasserman
 Winter Partners (OSMOSYS)
 Wirfs-Brock

What a long list. And there have been many deeply held views about which are "good" and "bad" – the "method wars" within the object oriented community, if you like.

Problems that can occur if you use some of the OO methods described above. Very often coding starts too early, partly because programmers feel happier writing code than papers, partly because managers feel things are going well if code's being written. Also partly because some of the methods don't translate too well into the "small matter of programming", and those in the know can start coding early to sidestep problems!

UML is an attempt to solve some of these problems, and has the potential to become the de-facto standard.

2.3 Unified Modelling Language (UML)

UML is used to model systems. It comprises element which are categorized as Views, Diagrams, Model Elements and General Mechanisms, and can be extended to include other elements such as Stereotypes, Tagged Values and Constraints.

A number of tools that use UML and help you implement the mass of graphics, design, inter-relationships, etc., exist. One example is Rational Rose, now an IBM product since IBM completed the purchase of the Rational company in early 2003. Such tools will comprise some (or all) Drawing Support, Model Repository, Navigation, Code Generation, Configuration, Version Control and other associated tools.

Let's look at the elements of UML

Views

A model comprises:

- The use-case view which describes the functionality that the system should deliver as perceived by the user of the system – the external actors. It's central to the system; the objective of the OO project as a whole is to provide the functionality of this view.
- The logical view describes how the functionality shown in the use-case view is provided; it describes the structures such as classes and objects, and also shows how they collaborate with each other. This view is primarily of use to the designers and developers.
- The component view is a description of each of the implementation modules, (i.e. sections of code; classes; clusters) primarily used by developers. It also includes ancillary information, like who's responsible for what.
- The concurrency view. Addresses issues that are of concern with multithreaded systems. Divides the system into processes and sections that can run in parallel, and deals with synchronisation of resources and what can be done asynchronously.

- The deployment view shows the physical layout of the system, the computers and other devices concerned, and which components are deployed onto which devices.

Diagrams

Each view comprises a number of diagrams.

- use-case diagram

The use-case diagram shows the external actors, and how they connect to the cases that the system provides. Our earlier example using a chapter object didn't actually have a use-case diagram, as we were talking about chapters in isolation from any real use of those chapters. But we do need such a diagram to create a reason for us writing the chapter in the first case!

Using our chapter class in a complete system, we might see something like Figure 7.

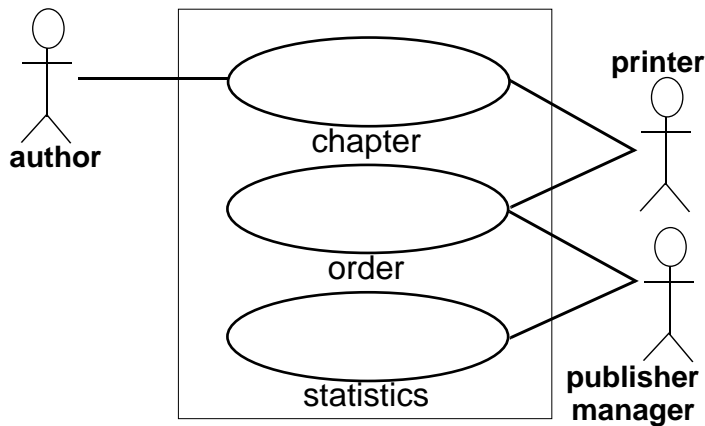


Figure 7 Our chapter class in a complete system

The author is purely concerned with the chapters. The printer is concerned with orders to print, and with the chapters; he needs to combine the two in order to undertake the functionality of his task. The publication manager isn't concerned with the actual chapter content; instead, he's concerned with orders for books, and order statistics.

- class diagram

A Class diagram shows how various classes relate to each other, and there are typically a number of class diagrams in a system. Very often, an individual class will occur on more than one class diagram.

Classes relate to each other in different ways. Conventionally, one class can extend another and then inherit from the base class.

Classes can be grouped into a cluster or a package. But also, Instances of one class can contain instances of another class.

An object of type chapter contains an object of type String and an object of type Author.

The new method (the constructor) for a chapter refers to 0 or 1 object of type File. Such an object of type File may be used by one or more objects of type chapter.

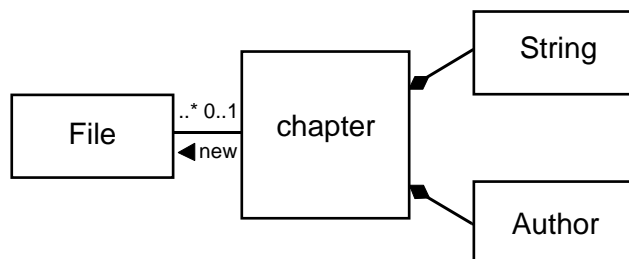


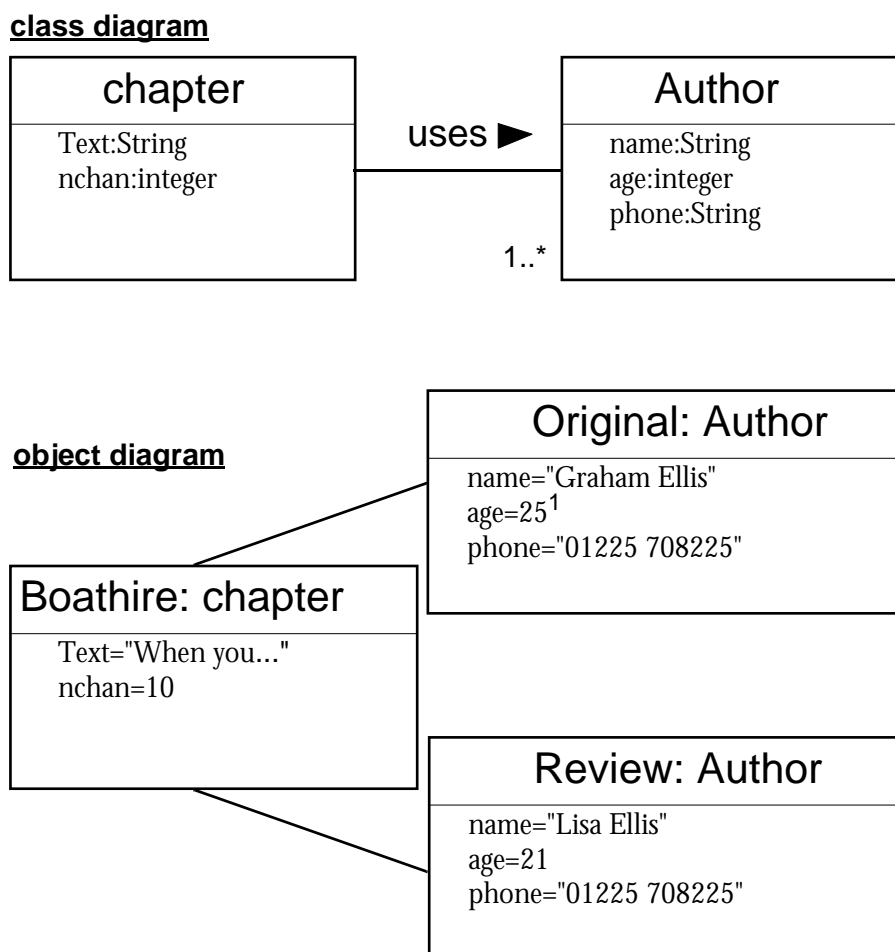
Figure 8 Partial class diagram for our chapter class

To this diagram, we could add our Tag and TagType objects as we developed our model as a whole ... and class diagrams can also include much more information such as names and types of primitives and system objects used within each class..

– Object diagram

An object diagram shows instances of a class, rather than the class as a whole. They can be used to clarify the design by showing a series of objects, or where a class contains multiple objects of another type. Our example *chapter* class contains a single Author. if we had it contain two authors, then we might have the class and object diagrams as shown in Figure 9.

Figure 9 class diagram, above, and object diagram, below, for our chapter class



It's no coincidence that these diagrams are similar to the informal diagrams we were drawing above. After all, UML is a modelling language, a way of specifying the topics that we were discussing in a formal language.

– State Diagram¹

UML's State diagrams are a formalised version of the state diagram we saw earlier in this section, Figure 4 and Figure 5.

– Sequence and collaboration Diagrams

A sequence diagram is used to show a dynamic collaboration between a number of objects.

¹ He lies.

Our chapter example would need considerable extension for us to find you an example, so shown in Figure 10 is an example of a sequence diagram (this one uses informal notation rather than the UML conventions, by the way).

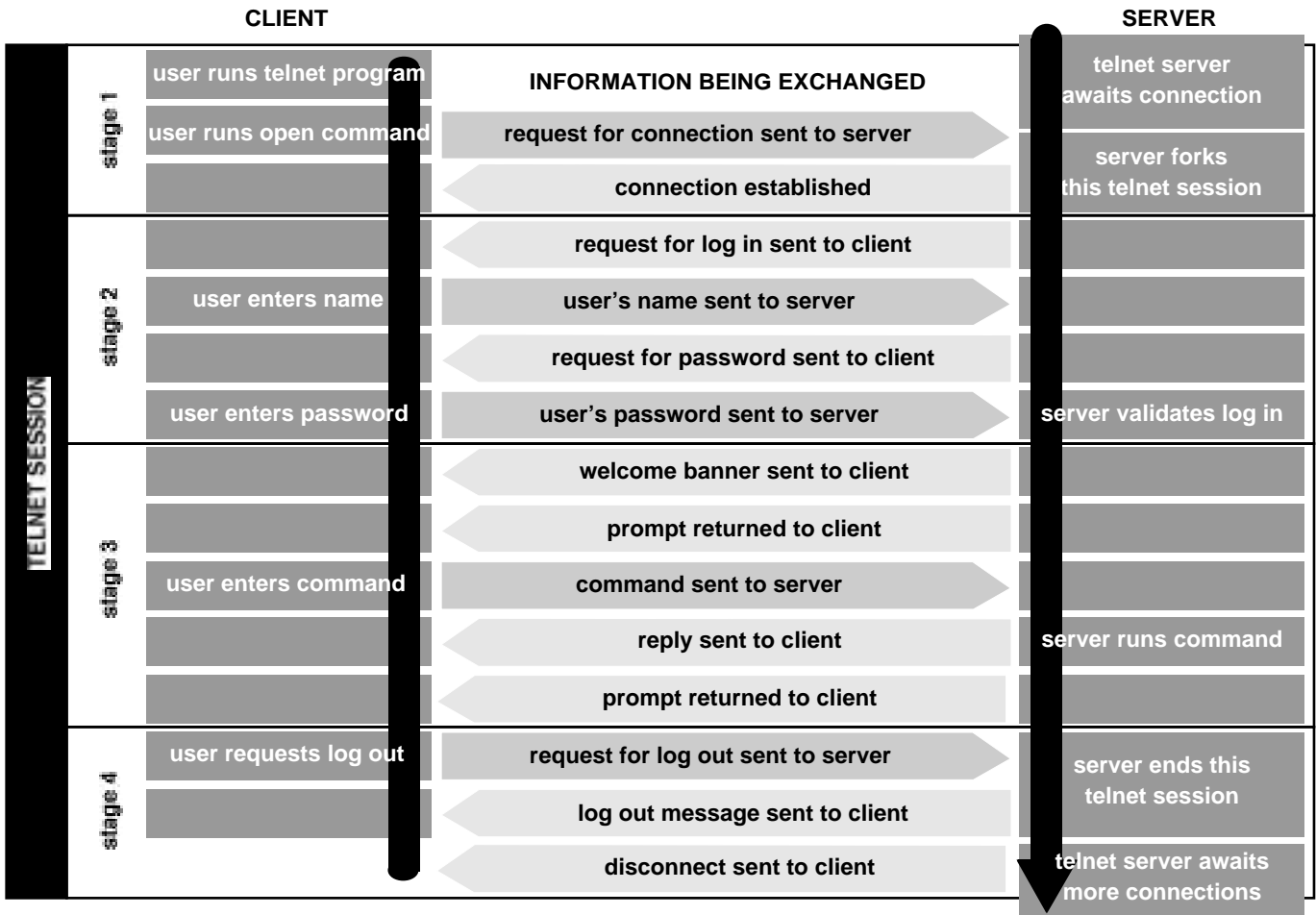


Figure 10 sequence diagram using informal notation

A collaboration diagram is often an alternative to a sequence diagram; it shows how things work together. If the timing is important, you'll use a sequence diagram. If it's the nature of the relationship between the two things collaborating, use a collaboration diagram.

- Activity Diagram

A formalised flowchart, but using a differing notation and terminology to the informal one we used earlier.

Figure 11 An activity diagram

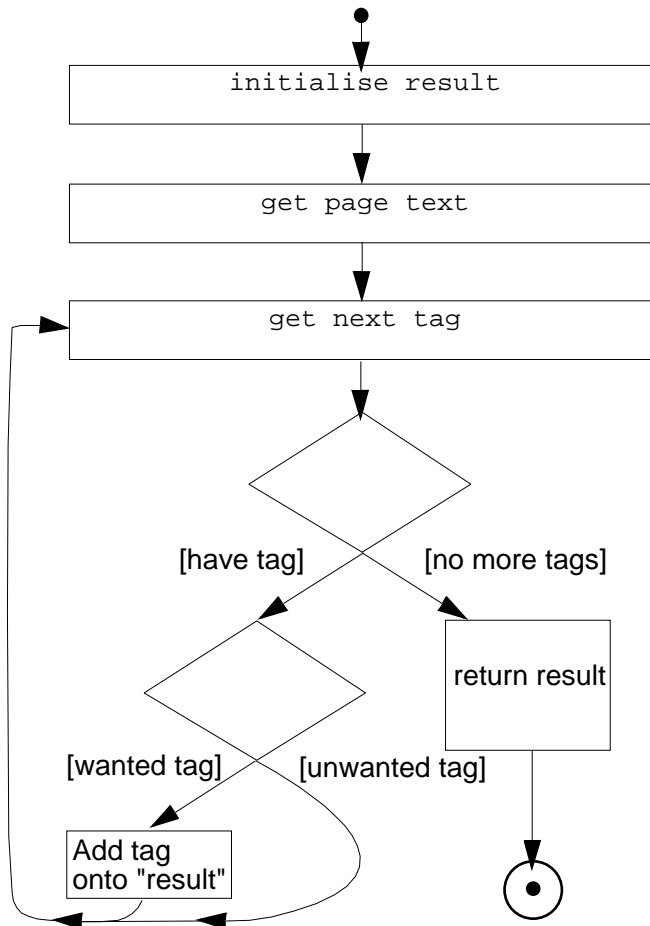
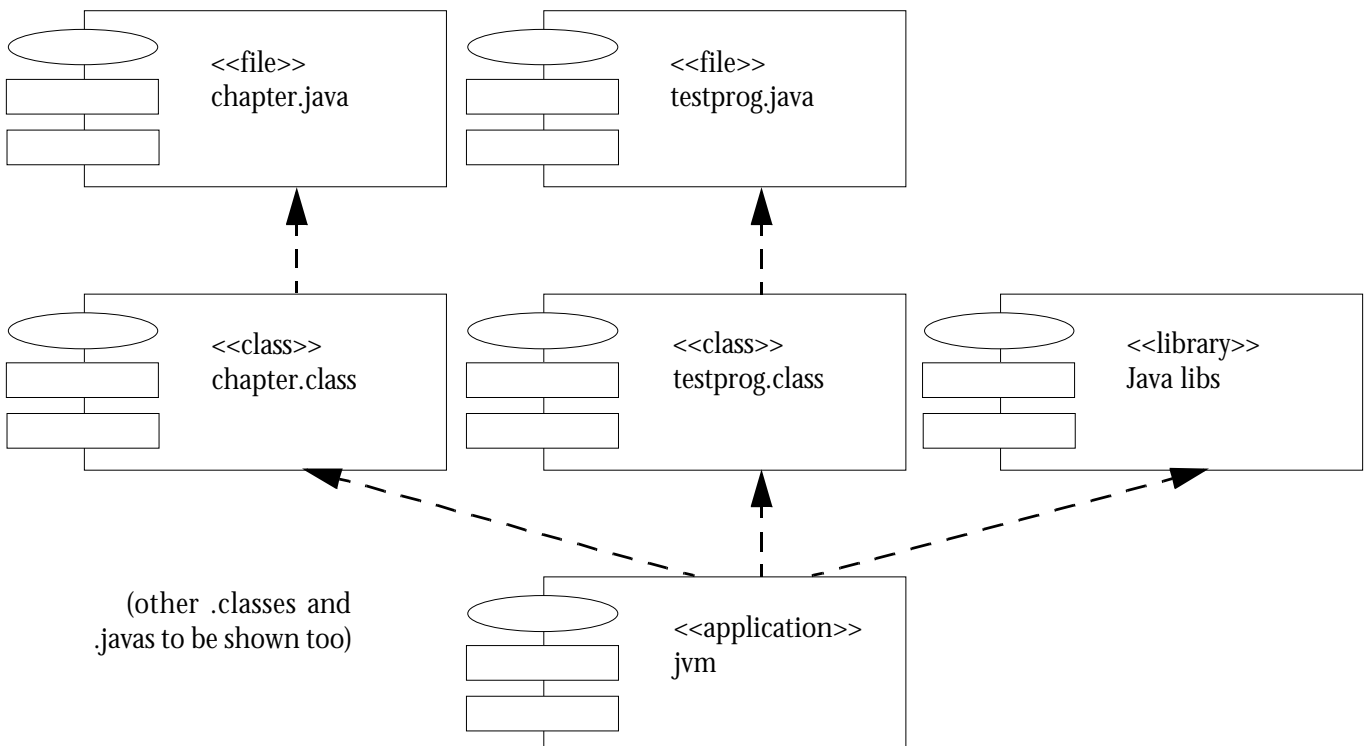


Figure 12 Our chapter class and test harness in Java

– Component Diagram

The layout of source, binary and executable files; shows dependencies, file types...



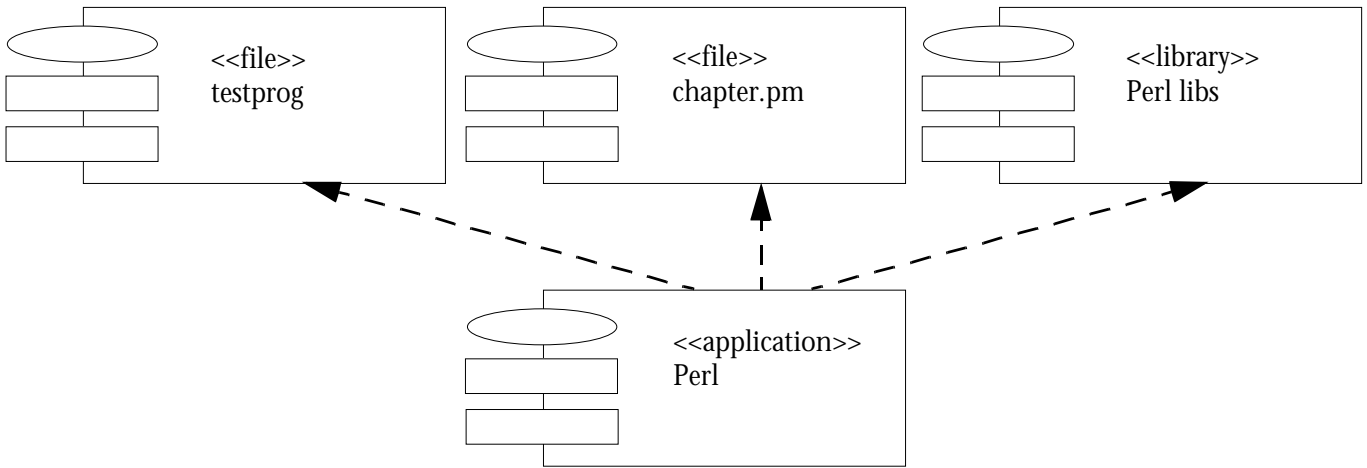


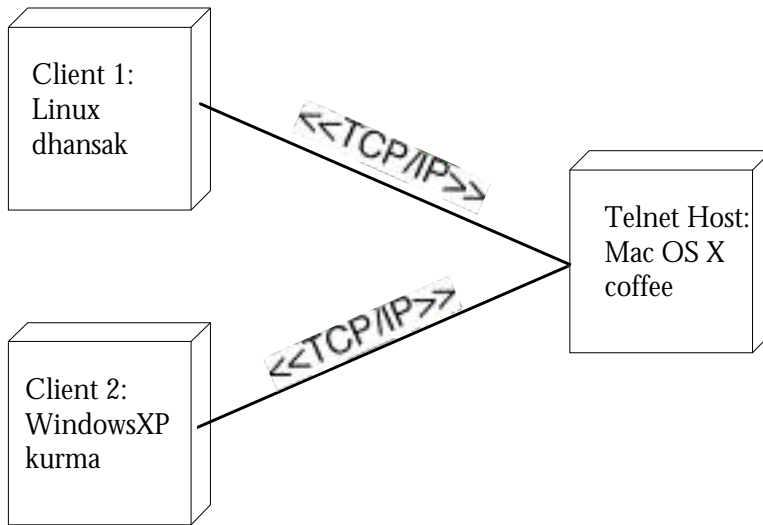
Figure 13 Our chapter class and test harness in Perl

Component diagrams often include documentation files too:
 <<page>> for HTML documents
 <<documentation>> for other forms of documentation

- Deployment Diagram

Physical architecture of hardware and software. In the case of our chapter class and test harness, everything will be running on the one computer and the diagram would comprise a single box. If we go the more complex use-case view we saw earlier, or to our telnet session that we showed in the sequence diagram, things can get to be more interesting, as shown in Figure 14.

Figure 14 A sequence diagram of our telnet session



Model Elements

We've seen a number of types diagrams, and they are related. Model elements are shared between and used on multiple diagrams of different types, but there are rules concerning which elements can be used on which diagrams.

As well as the formal model elements, there are a number of general mechanisms which can be used in all diagrams. These general mechanisms are subdivided into types such as adornments, notes and specification.

UML can be extended or adapted by the addition of further facilities, such as stereotypes (a mechanism that allows a modelling element to be based on an existing modelling element), tagged values (where properties are associated with an element)

and constraints (where restrictions are placed on values that are held in elements).

UML Summary

UML is just a way of formalising design and avoiding design pitfalls; whilst it's a help, you still need to be aware of overall design principles, of what your target language is capable of, and of what your user requires in the end!

2.4 Tools

UML is a tool that's designed to assist in the formalising of design over a number of designers. It certainly makes the designer think through his design, as you'll appreciate from what we've already covered in this module. But how practical is it?

For a simple system, diagrams can be drawn and correlated easily. But no system is that simple and with a series of diagrams to describe your system, a change will mean going through a pile of papers. There must be an easier way of doing it.

Yes, there are commercial tools available which let you define your model as a whole and then extract individual views and diagrams as required. Rational Rose¹ is perhaps the best known.

What's offered by such tools?

- Drawing support
- Model Repository
- Navigation
- Code Generation

And going beyond the the strict implementation of UML, there are many other facilities available in such tools:

- Configuration,
- Version Control,
- Prototyping tools
- Testbed tools (e.g. beans)

2.5 Project management and design issues

In this module, we've looked at some of the fundamental concepts of Analysis and Design of object oriented systems, in preparation for programming those systems in Object capable languages such as Java or Perl.

As well as informal techniques, which we studied so that you're aware of the basics, we had a brief look at UML, which lets you add a formality to the design process; in turn leading to advantages if you're automating the design through design tools, or if you have a project task.

Software design, though, is not easy and it needs careful planning and management. You need to get it right, you need to look ahead, you need to assess any risks you're taking and plan ahead of time for them.

Through the life of the project, you need to keep one eye on your goal and make sure you're not going off track, and one ear open to your user community to ensure that your implementation is fit for the purpose.

Two quotes from Grady Booch, Chief Software Engineer at Rational and often regarded as one of the fathers of OO Design:

"Remember that the class is a necessary but insufficient means of decomposition. For all but the most trivial systems, it is necessary to architect the system not in terms of individual classes, but in the form of clusters of classes."

"To be successful, an object-oriented project must craft an architecture which is both coherent and resilient and must then propagate and evolve the vision of this architecture to the entire development team."

¹ Company – Rational (now part of IBM); product – Rose

2.6 Extreme programming

Extreme programming is a design and programming methodology that's becoming increasingly popular. Its unit coding and unit programming approach fits well with an object oriented approach, but in extreme programming you're going to be encouraged to redesign and refactor the objects as the project progresses. And you're encouraged to involve the customer on day 1, and day 2 and every other day too.

Here are some of the keys ...

- User stories. Typically customer requirements of just a few sentences each, taking a couple of weeks each to complete. There may be between 50 and 100 user stories in a typical XP project
- Spike Solutions. Not wasting a lot of time writing code that's not effective in the end, and experimenting and trying out code in such a way that there's no huge loss if it has to be junked, then
- Refactoring. Redesigning or altering aspects in the light of experience. You are encouraged to refactor early and often, and not continue on down a less than ideal development route.
- Unit tests. Testing each element of the system in turn to see how it works before integration to ensure a series of robust components.
- Frequent Sequential Integration. Adding the units together frequently to ensure that they work correctly together, and adding them one at a time.
- Acceptance tests. Ensuring that each user story is met to the satisfaction of the user through rigorous testing.
- Iteration and release plans. A series of integrations and releases as the work proceeds, with feedback.

Planning

- User stories are written.
- Release planning creates the schedule.
- Make frequent small releases.
- The Project Velocity is measured.
- The project is divided into iterations.
- Iteration planning starts each iteration.
- Move people around.
- A stand-up meeting starts each day.
- Fix XP when it breaks

Design

- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible

Coding

- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All production code is pair programmed.
- Only one pair integrates code at a time.
- Integrate often.
- Use collective code ownership.
- Leave optimization till last.
- No overtime.

Testing

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.



Exercise

You've got a database of golf courses containing the name of each course and a series of numbers – the length and par of each hole. Your application is to write a piece of code that goes through each of the golf courses in the database and reports back on the par and total length of each course.

Produce a series of diagrams (use case, object, class, activity, component, deployment) for this scenario. If useful, produce sequence and state diagrams.

License

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log
Original Version, Well House Consultants, 2004

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

Updated by: _____ on _____

License Ends.