PIY-II and Extreme Programming

David Vivash

2nd May 2001

Project Supervisor: Mike Holcombe

Abstract

There are many different methodologies that can be followed to produce a piece of software, some of which are easier than others to follow. This project has looked at one specific lightweight methodology created by Kent Beck known as *Extreme Programming*, or XP.

Throughout this project, the various shortcomings of XP are highlighted and possible solutions are offered. Specifically, attempts have been made to tighten the definitions of terms invented within the XP world but never succinctly explained, as well as an attempt to bind the gap between the user stories (initial system specification) and the final software. It is hoped that in binding this gap, more hope can be offered to the software developer wanting to produce correct software.

The analysis of XP is based on experience gained whilst applying it to produce a particular piece of software, known as PIY-II. PIY-II is the successor to PIY, the product of a previous dissertation project. PIY, *(Program It Yourself),* is a tool aimed primarily at non-programmers to allow the creation of applications without the need for code. By using XP practices, PIY-II has been written with a great deal of success, and has grown into a fully extensible software development environment.

Acknowledgements

Thanks firstly to my project supervisor Mike Holcombe.

Thanks to the XP mailing list, especially Ron Jeffries, Martin Fowler and Michael Feathers, who probably do not know how much they helped me to understand the underlying concepts of XP.

Thanks also to my house mates Amy, Steve and James, who managed to seem completely at ease with rising piles of washing up.

Finally, thanks to my nieces Carinda and Leona, who tried their hardest not to play with my computer over Easter when the majority of this dissertation was written up.

Chapter 1: Introduction

This project is a follow up to work done on PIY in previous years. The difference this time around is that *Extreme Programming* (XP) has been used for development. This new software design methodology has gained a lot of press recently, and a number of common misconceptions persist as to what it actually is.

Before I explore either XP or PIY though, I will briefly comment on the main goal underlying this project - namely to analyse whether XP really does help development teams to deliver *correct* software. This is the goal of all methodologies, so does XP achieve it?

[HOLCOMBE98] points out that the exact meaning of a system being correct differs depending on who you ask - a user will consider a system correct if it solves the problem they need solving, and doesn't crash. An academic computer scientist will consider a system correct if it can be formally proven to satisfy the mathematical formula that defines it.

Formally defining a system is much more difficult than actually designing the system - most software design methodologies rely on testing their systems rather than proving them correct. This approach has disadvantages, as any undergraduate computer science student will tell you that "testing cannot prove the absence of bugs". However, can proofs prove the absence of bugs? How do we know the proof is correct? Is it not true that a proof is correct only when it does enough to convince a human reader that it is correct, a nobody else can prove it incorrect? Many software engineers prefer simply to test, and take this as being all of the evidence for correctness that they need. A famous Donald Knuth quote is "Beware of bugs in the above code; I have only proved it correct, not tried it." XP attempts to satisfy the user that the system is correct, not the computer scientist. Correctness is a term seldom used in XP circles - customer satisfaction is considered much more important.

So in using XP to develop PIY-II, I will be evaluating XP effectiveness at creating a correct system. It is, of course, dangerous to base opinions of a methodology on how well it works on a small project - methodologies show their real worth when applied to large scale projects. On a small system like PIY-II, competence and force of will are often sufficient qualities needed to succeed, regardless of the methodology used.

To summarize, this project aims to fulfill two main goals. These are:

1. Create PIY-II, the successor to PIY.
2. Critically assess *Extreme Programming*.

Chapter 2: Introduction to Extreme Programming

Writing a piece of software is easy -
you just think of what you want the software to do, and then code it.

Most software developers would laugh at the above statement. But why? Anyone can decide what some software should do, and programmers can do the coding. Most software developers would laugh at that statement too. The reasons for this are numerous, but often include:

1. Customers don't know what they want the software to do - they know how to describe the problem, but want the software to provide the solution. They don't actually know the solution. Sometimes they're not even that sure of the problem.
2. Programmers can produce pieces of code, but any arbitrarily complex piece of software consists of many pieces of code interacting in many different ways. No software programmer is going to be able to store a whole software system in their head at any one time - the system needs to be reduced into manageable chunks.

It is in solving these kinds of problems that software methodologies arose. That is, software developers wanted a set of rules they could follow to make the transition from the customer's vague idea of what a software solution should do, to a finished system. Over time they realised they wanted their methodologies to do more. They wanted to be able to reuse elements from one project to another, they wanted to be able to produce software that was easily maintainable, and some of them even wanted their customers to formally specify their systems for them… This momentum of continually adding to simple methodologies resulted in what have been dubbed "heavyweight" methodologies.

2.1: Heavyweight Methodologies

Heavyweight methodologies have a reputation for three things: bureaucracy, being unpopular, and not actually working. We could have learned to live with the first two things, but the third is a bit of a stumbling block. It's not entirely true - by following a heavyweight methodology you *can* produce a piece of software, but often at the cost of slowing the development pace down to an unacceptable level. For many projects this is too high a price to pay.

2.2: Lightweight Methodologies

So rather than going forward and adding more features to old methodologies, some software developers went backwards. By collecting together the practices that actually worked from the heavyweight methodologies, a new breed of methodologies were formed: "lightweight" methodologies. From distance the lightweight methodologies look like the heavyweight methodologies with most of the documentation removed. This view is partially correct - many lightweight methodologies take the view that the key part of documentation is the source code. However, as Martin Fowler points out, this lack of documentation stems from two key concepts in lightweight methodologies:

1. *Lightweight methodologies are adaptive rather than predictive*. Since customers change their minds, rather than predict changes and plan the system with predictive

measure in-built as many heavyweight methodologies do, lightweight methodologies are designed to produce systems which can adapt to the change when it occurs. (For an example of this later, see YAGNI in XP).

2. *Lightweight methodologies are people oriented*. Rather than design the methodology around the *process* of designing software, lightweight methodologies concentrate on the people involved - the relationship between the customer and the software developer.

These things are all well and good, so long as they work. So do they? The purpose of this project is to look at a relatively new lightweight methodology known as Extreme Programming (XP). There are many other lightweight methodologies, such as Feature Driven Development, SCRUM and dX (an minimal implementation of the Rational Unified Process), so any conclusions I draw from my study of XP may not be representative of lightweight methodologies in general. In particular, this project is an attempt to ascertain the effectiveness of XP in producing "correct" software, and to see whether XP would gain by enforcing a more formally structured framework for it.

Chapter 3: Introduction to PIY-II

In order to analyse the performance of XP, a piece of software will be produced using XP practices. The software in question is *Program It Yourself II* (PIY-II) - a sequel to the original PIY, which was a piece of software developed in a previous dissertation project.

3.1: Previous PIY implementations

In effect, PIY-II is to be a tool which will enable non-programmers to create their own software. That is, any average computer user should be able to construct their own applications without writing a line of code. As mentioned, PIY-II is the sequel to a previous PIY project - hence a lot of research into what is feasible for PIY to include has already been performed. The act of actually gathering requirements has therefore not been performed since the information is already available - PIY-II will have a similar philosophy as Phil McMinn's PIY project, but will attempt to overcome its shortcomings.

One problem I saw with McMinn's project was the difficulty in adding support for new types of buttons into the PIY environment. I decided that PIY-II would be as flexible as I could possibly make it - a plug-in architecture should be built to allow new components to be added to PIY-II with ease.

Another problem I saw with McMinns project was that users had very little control over the logic behind the user components they added to their GUI's. For PIY-II I decided to add the ability for users to add a sequence of actions to be executed once a component event occurs. That is, users can specify that a particular button click will quit their application, a different button click could load in a text file etc. Again, I wanted the possible actions to be as flexible as possible - I wanted my plug -in architecture to extend to actions as well as components.

In essence, this sums up the overall operation of PIY-II. Users can create a user interface, and from there they can specify what should happen on each button click, and each window resize, etc. One further development of PIY-II was to allow the compilation of projects. Users can create a project in PIY-II, compile it, then run their compiled application on any Java-enabled system.

3.2: Commercially Available Products

There are surprisingly few commercially available products which enable the creation of applications without the need for code. Indeed, if an average computer user wishes to create their own programs, the only real option available to them is to learn a programming language.

3.2.1: Fourth Generation Languages

Fourth Generation Languages (4GLs), such as Visual Basic, JBuilder and Visual C++, make the construction of graphical user interfaces (GUI's) a much simpler process. However, the user of these programs is required to attach code to buttons, for example, to determine what happens when the button is clicked. Thus knowledge of the principle programming language the 4GL is based on is required to generate all but the simplest of applications.

For creating a GUI though, the currently available 4GLs are the best in the field. This makes 4GLs very quick at creating prototype systems which just show a GUI, to give the client some idea of what the finished product will look and navigate like.

3.2.2: Visual Programming Languages

Visual Programming Languages (VPLs) allow the user to create whole programs without the use of any code. This is essentially what PIY-II will attempt to do, and so if it is to be successful, it's important to see whether the approach will have any benefits over using 4GLs. There's no point somebody learning to use a VPL well and being limited in what they can do in it, if it would have taken them a little more work and produce something better in a 4GL. The important thing about a VPL is that it *must be much easier to create a simple application in than a comparable application in a 4GL*.

It would be naive to assume that a VPL is limited in what it can do just because no code is written. There's no reason a VPL couldn't offer enough functions to the user to enable them to do whatever a 4GL user could do. It seems the approach is coming from the other direction though - 4GLs are gradually turning themselves into VPLs. A lot of commercially available 4GLs allow their users to write simple applications with hardly any code at all. Indeed, by just loading and compiling some of the example programs supplied with some 4GLs reveal that writing simple text editors and graphics viewers require very little code. It's unlikely any of the 4GLs mentioned will ever actually reach the stage of requiring no code, but it's important to see that PIY-II will only be considered successful if it can create these simple applications as simply as the 4GLs.

3.2.3: Java Studio

The only major VPL is an application produced by Sun Microsystems called *Java Studio*, although development of the package has now ceased. No code is needed to write a Java Studio application - everything is produced by connecting components together in a design window. Java Studio's main purpose is to allow the easy construction of applets. Its main problem is that the entire logic for the program is displayed in a single window. This makes it hard to follow for large applications. Also, it is clumsy in some tasks - such as trying to implement basic mathematical formulae, and retrieving text from text input boxes (the user of the application is expected to press enter after entering data into a text box - pressing tab to move onto the next text box won't work). With the above in mind, and taking into account points raised in McMinn's project [MCMINN98], the Java Studio approach will not be emulated in PIY-II.

3.3: PIY-II

To reiterate, PIY-II is to be a tool which will enable non-programmers to create their own software. The purpose of this dissertation is to see how well the development of PIY-II has been supported by the XP practices.

Chapter 4: Extreme Programming in detail

XP is the name Kent Beck has given to a lightweight design methodology for creating (generally business) applications. It is made up of 12 explicit practices:

1. Planning Game
2. Frequent Releases
3. System Metaphor
4. Simple Design
5. Unit Tests and Functional Tests
6. Refactor
7. Pair Programming
8. Collective Code Ownership
9. Continuous Integration
10. Forty Hour Week
11. Onsite Customer
12. Coding Standards

There are a few other practices that don't seem to make an appearance in this original list, specifically the "test-first design" practice. This practice will be considered with practice 5 (unit tests and functional tests).

4.1: The XP Practices

(Note: The glossary in appendix A details many of the terms used within the descriptions of each of these practices).

4.1.1: Planning Game

This occurs at the start of an iteration. Here, the customer presents the programmers with some stories to implement in the next iteration. The programmers estimate the amount of time it will take to implement each story in ideal programming days. The customer then decides which of the stories will add the most business value to the product in light of the time it will take to implement the story, and selects the stories from the initial set that (s)he wishes the programmers to implement. The estimation of the time a story takes to implement is based on the time it has taken stories to be implemented in the past (i.e. estimation by comparison). [JEFFRIES00] contains a complete and very readable description of how to estimate story lengths in XP.

Now it's up to the programmers. Each story is split into engineering tasks - ie. the things that need to be programmed to get the story working. Each programmer can sign up for different engineering tasks, estimating the amount of time it will take to implement the task. The total time they estimate for all the tasks should be equal to the time of a single iteration. Note that two programmers will sign up for any one task, see pair programming.

4.1.2: Frequent Releases

Release early, release often. The "release" is not a *demo* version, it's an *actual* version that does something useful. It's not always immediately obvious how to create a program

encompassing enough functionality to be useful - some systems really do seem to be all or nothing - but more often, with a little thought, useful products can be delivered iteratively. Again, see [JEFRRIES00] for demonstrations of this concept.

4.1.3: System Metaphor

The system metaphor is a set of classes an patterns that make up the core of the system. A primitive architecture, in a way. However, this definition is certainly not standard. I am yet to find a standard definition of the term "metaphor", so this definition will have to do.

The definition I have given is close to the [XPMAP00] site, which describes the system metaphor as "The Extreme Programming (XP) word for architecture. A handful of classes and patterns that shape the core flow of the business the system will implement." The subsequent discussion of metaphor on the site, however, tends to imply that the metaphor exists as a form of communication between the development team and the customer - it's a way of "thinking" about the system.

This portrayal of what the system metaphor should be is repeated in other pieces of XP literature. [JEFFRIES00], for instance, describes the metaphor in this way:

> *" ...the idea that each application should have a conceptual integrity based on a simple metaphor that everyone understands, explaining the essence of how the program works. The essence of this idea is to have a common concept of what the program is 'like'".*

I have seen many other interpretations of what a metaphor really is. In one extreme, there is the idea that the metaphor is set down in some class diagram representing the solid core of the system, the other extreme is the idea that it just some abstract description of the system which can be used for communication.

The pure XP practitioners who avoid any form of an up-front design session tend to stick the Jeffries definition of metaphor. Most others start with a class diagram as a metaphor, and work from there. This is the approach I have taken.

4.1.4: Simple Design

The name of the practice says it all - keep your design simple. XP defines a simple design as one which satisfies the following properties:

1. Runs all the tests;
2. Expresses every idea you need to express;
3. Contains no duplicate code;
4. Has the minimum number of classes and methods.

Two further terms which are used regularly to remind XP programmers to design "simply" are "Do the simplest thing that could possibly work" (DTSTTCPW) and "You aren't gonna need it" (YAGNI).

YAGNI, in particular, is something which is often misinterpreted, and is not as easy to adhere to as one might imagine. The philosophy behind the term is that a programmer should only implement methods which he needs to implement to get a particular test to pass. Superfluous methods are a waste of time, no matter what justification is given. Maintaining code that is not used is a waste of time. YAGNI follows directly from the principle of simple design. The reason this can be difficult to adhere to is because of design patterns (primarily). A design pattern specifies the classes and methods needed to implement it - but are all of these methods *needed*? Applying YAGNI seems to suggest that design patterns should be modified to suit the design problem. This seems reasonable, except the design patterns literature will point out that the pattern has been proven to work in its generic form on a number of successful and well designed systems - why modify something that's proven to work? My personal view is to stick to YAGNI even in these cases. Extra methods can be added as needed, whereas extra methods that are added but never used cost time in the initial implementation and testing, and in subsequent maintenance.

That is, code what you need, not what you think you might need.

4.1.5: Unit Tests and Functional Tests

The first thing to note about XP tests is that they must be automated. At any time, the system should be in a state whereby it can be run through the set of tests.

A unit test is, in its simplest form, a test of a single method. The test case might send particular values to the method, and check that the results returned (or observed) are indeed the results expected. Unit testing is very well supported in most programming languages. One facet of unit testing not explicitly mentioned in the original list of 12 practices is the practice of *test-first design*. That is, before a method is written, a unit test is written for the method. Upon running the test, it fails (you haven't coded the method yet). You then go ahead and attempt to write the method. You then run your test again, to see if the method works. If it doesn't, you go back and fix the errors then run the test again. You continue to do this until the method does everything necessary to pass your tests. This process ensures that a complete set of tests is always available for the system - indeed, the tests must all pass 100% of the time.

The construction and management of these unit tests can be done with the xUnit series - unit testing frameworks which are (freely) available for many programming languages, from Smalltalk to Visual Basic. The consequence of having a unit testing framework in place is that your code always has a complete set of tests associated with it, and the fact that they're all automated means that bugs in any part of the code can be quickly detected whenever a change is made.

To complement the unit tests which test individual classes in isolation, the functional tests (acceptance tests) are owned by the customer and test functional requirements of the system as a whole. It is up to the customer to define what functionality the system needs, and in designing tests for these requirements ensure that they are met. In contrast to the unit tests which must pass 100% of the time, it is not essential that the functional tests do. Whilst it is certainly desirable to implement all of the required functionality, time and budget constraints often mean that not all of the requirements get implemented. It is the role of the planning

game to identify which requirements will offer the most value to the customer, so that these requirements can be concentrated on.

4.1.6: Refactoring

Refactoring is the process of restructuring code without changing its function. The purpose of this is to help accommodate the XP practice of simple design. Refactored code should be simpler and easier to understand than unrefactored code. [FOWLER98] gives a thorough treatment of refactoring.

It is perhaps worth mentioning the impact that refactoring could have on programming in general. The term *refactoring* was first coined by Opdyke in 1992 (see OPDYKE92), but XP is the first methodology to rely solely on refactorings to guide the design. This concept of emergent design via refactoring is starting to gain more usage outside of XP circles. A major step forward in refactoring came when a *refactoring browser* was launched for Smalltalk. A refactoring browser is essentially an IDE in which simple refactorings can be carried out with the click of a button.

Since the Smalltalk browser, there have started to appear a few refactoring browsers for Java. These browsers generally incorporate themselves into a specific existing IDE (such as IBM's Visual Age, or Borland's JBuilder) to add refactoring capabilities. An article by Martin Fowler explains that these tools supply more than simple "cut-and-paste" refactorings. A refactoring such as "Extract Method" results in a selected block of code being relocated to its own method, being replaced by an invocation of that method. This has consequences in sorting out the scope of the variables required by the block of code, so that the new method has access to these and still works properly. Much more complicated than simply moving some code blindly - this refactoring requires that the structure and meaning of the code is considered. Now that there exist tools to automatically perform refactorings (which Martin Fowler terms "crossing the refactoring rubicon"), it is very likely these tools will become standard use outside of XP circles.

There are hundreds of useful refactorings that can be applied to code, most of which can be performed mechanically. I predict that the future will bring general refactoring to most major IDEs and programming languages on the market, making refactoring a common part of coding no matter the methodology being used, and making those IDEs more invaluable.

4.1.7: Pair Programming

Pair programming is when two programmers work together to produce code. That is, one of the programmers is at the keyboard typing (he is known as the *driver*), and the other programmer watches to spot errors and to solve any design or implementation issues there might be.

As has been mentioned, the practices in XP are not new. Pair programming has been around for a while and has been the subject of much successful research, such as [WILLIAMS00].

The common result of pair programming research is this:

Pair programming works.

There are many programmers who can't believe that working with somebody else can improve their code - programmers who have enormous confidence in their own programming ability. There are programmers who don't want to spend time teaching less abled programmers, and programmers who don't want to be slowed down by a pair telling them to where to indent their code. Research carried out, however, suggests that virtually all programmers gain from pair programming.

4.1.8: Collective Code Ownership

Nobody "owns" a particular class - if somebody needs an extra method in a class they can go ahead and add it themselves. If the programmer didn't own the class they'd need to find the owner, ask them to add the method, then wait.

Once the owner had implemented the method, the programmer could get back to what it was they were doing and make use of the method. Of course, it wouldn't quite happen this way. Firstly, the owner probably wouldn't have implemented the method to do what the programmer wanted exactly, and secondly the owner would have taken too long to add the method, and the programmer will have worked out a workaround to the problem which doesn't need the method. So the method becomes redundant but needs to be maintained.

So in not having collective code ownership, a number of other XP practices would be implicitly broken, not least simple design and refactoring. (Refactoring involves removing redundant or duplicate code - something we can't actually do if we don't "own" the code).

4.1.9: Continuous Integration

Having many programmers working on different parts of the system at the same time causes many integration problems. That is, when a particular module is integrated into the system after weeks worth of work has been done on it, the task of actually incorporating the new module into the existing codebase takes a lot of time and effort. The XP solution is to integrate all of the time. That is, code should be integrated into the system at least a few times every day. There is one prerequisite to integrating new code, and that is that all of the unit tests for the code must pass 100%. If they don't, the code is not allowed into the system

Once all unit tests pass, the code should be reintegrated immediately so that all programmers are working on the most up-to-date version of the system.

4.1.10: Forty Hour Week

Tired programmers make more mistakes. Do not work more than one consecutive week of overtime. This advice could be applied to many other industries - it's not an excuse to be lazy, it's an attempt to keep quality at an acceptable level.

4.1.11: Onsite Customer

This seems to be a very difficult practice to adhere to, but since there is no real requirements analysis phase in XP it turns out to be a very important practice. The idea is that a representative of the customer is available on-site where the programmers are working. XP advocates recommend that programmers all work in the same room, two programmers per

computer. They also recommend that the customer have a small office set up in the corner of this room where they continue with their normal job. The idea behind this is that whenever a programmer has a query about a particular story they are implementing, they can immediately ask the customer. Face to face communication is what XP values the most - communication over the phone, or even worse, via emails, is a weak substitute.

Ron Jeffries remarks that in his experience of working in XP teams, even if the customer is only a few floors away, programmers will put off contacting them for trivial problems and will just guess "for now", and make a note to ask the customer later. Not having an on-site customer will invariably increase the number of guesses made in the code, even with pair programming.

Note: the original XP project - the Chrysler C3 project - failed because it did not adhere tightly enough to this rule. See chapter 6 for more details.

4.1.12: Coding Standards

The coding standards in a project simply define how code should be structured in terms of capitalisation, indentation, bracketing, etc. The coding standards should also be in place to ensure consistent class and method naming, to help with refactoring and collective code ownership. It is also partly a defense mechanism for pair programming - the programmer does not need to worry that their pair will be concentrating on their code layout style, since everybody should be using a pre-agreed standard. Such a standard might be the Java look and feel guidelines, for example.

4.2: Brief Analysis of XP

XP is not new. However, the specific *collection* of practices is new. A lot of experienced programmers may look at the list and say *"Interesting. But I do these things anyway."* - XP is just common sense. But common sense doesn't make it a methodology, no matter how "lightweight". So what makes XP more than just common sense? In the words of Kent Beck, "XP turns all of the knobs to ten".

That is, if testing is good, do it all the time. If pair programming is good, do it all the time. If short iterations and frequent releases are good, make iterations as short as possible and release a system to the customer as soon as it offers business value. If simple design is good, refactor your code to something more simple whenever you have the chance. And so on.

There is a process behind the common sense (ie. the planning game), but one of the things that holds it all together is the practice of unit testing. Without unit tests, refactoring would not be possible, continuous integration wouldn't work, collective code ownership would become much more difficult, and it would be hard to stick to a forty hour week. In short, quality would be nonexistent and there would be no final product to release.

One other thing that holds XP together is pair programming. Since XP insists on doing pair programming all the time, it effectively becomes continuous peer review - an extremely effective quality assurance technique. Combining test-first design with pair programming seems, on the face of it, to be the core of XP quality assurance.

Chapter 5: PIY-II in detail

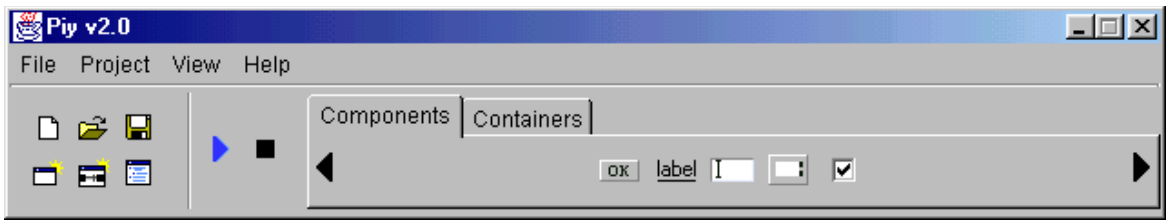
The goal of PIY-II was to create a flexible visual Java application environment which can be used to compile applications without the need for code. The only criteria I set for deciding whether PIY-II was successful was to see whether this goal has been met. That is, can Java applications be compiled within PIY-II without the need for code.

One point I think should make before I describe PIY-II is that I did not perform *any* requirements analysis, nor did I perform any modelling of the problem domain or consider alternative solutions. The main reason for this was because PIY has been done before: PIY-II is not new. There have been three previous PIY projects, all of which look into the feasibility of different solutions. McMinn's project also has extensive user tests, which analyse how easy his version of PIY is to use. Whilst the approach he took is different to the approach of PIY-II, much of the analysis of what users require of a PIY system is relevant to PIY-II.

Another reason for avoiding modelling the problem domain was due to the use of XP as a development methodology. The main premise of XP is that software engineering should be concentrate on producing code - there is a very low emphasis placed on the creation of other artifacts. The *initial* user stories (requirements) of PIY-II are provided in Appendix C, and the initial functional tests are in Appendix D. These are the only non-code artifacts created for PIY-II (except the system metaphor, which is described in detail in chapter 7).

5.1: The User Interface

The PIY-II user interface is made up of four windows. The main window is the project control window, shown below.



This is the window that is used for general project management such as loading, saving, running and compiling. It is also used to add new windows to the project and to add new components and containers to those windows.

5.1.1: Adding Components to PIY-II

New components can be added to PIY-II (automatically added to the toolbar above) by simply placing their class files in the "piy/usercomponent" package directory. The same is true for adding containers to PIY-II - just add their class files to the "piy/usercontainer" package..

The specific process by which PIY-II detects components is as follows:

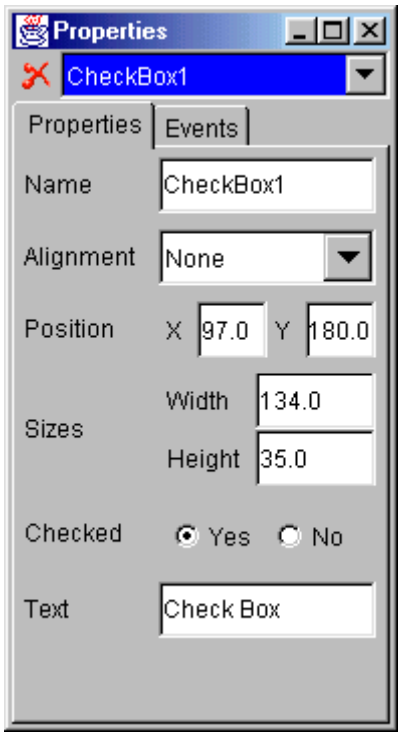
1. Firstly, the list of files in the "piy/usercomponent" directory is obtained. This list is pruned to include only those files ending in ".class".
2. From the list, an array of classes that implement the `UserComponent` interface is constructed.

3. Then an array of classes that extend the `Descriptor` class is constructed. These classes provide specific information about particular `UserComponents` for PIY-II's use, such as the component's name, and what icon to display on the toolbar.
4. Each `Descriptor` is matched to the `UserComponent` it is describing. If a `UserComponent` doesn't have a relevant `Descriptor` class, a default `Descriptor` is constructed for use.
5. PIY-II now has an array of descriptors that can be used to retrieve all of the information needed to provide components to the user.

The same process is used for retrieving `UserContainer` classes.

5.2: The Property Editor

The second important window in the PIY-II user interface is the PropertyEditor window, shown below. The main purpose of this window is to enable to user the edit the properties of the currently selected component, container or window.



The properties displayed in the window have immediate effect; when the user starts editing the title of the window in the edit box, for instance, the effect can immediately be seen on the actual window title bar.

The process by which PIY-II introspects an object to determine what properties are available to the user is similar in a way to the operation of *Java Beans* ™. In providing a new `UserComponent` for PIY-II, no real work has to be done by the component vendor - it is up to PIY-II to analyse the component and deduce what properties it can make available.

There are two main elements that make up a "property", which are

1. The property name;
2. The property type (ie. whether it's a Color, or a String, or an int, etc.)

The first thing PIY-II does to obtain a valid property from a component is to see whether there are two methods of the form

```
public void set<PropertyName> (<PropertyType> x) { … }
public <PropertyType> get<PropertyName> ( ) { … }
```

So if a component wishes to offer a "Background" property of type "Color", it might have two methods of the form

```
public void setBackground(Color x) { … }
public Color getBackground ( ) { … }
```

There is one further requirement for a property to be fully recognized by PIY-II, which is that the property type must be supported by PIY-II. The Color type is one such property type supported by PIY-II, but as with much else of PIY-II, this mechanism is extensible. That is, support for new property types can be offered via a plug-in mechanism. On startup, PIY-II detects all of the property support classes from the "piy/support" directory, in much the same way it detects new user components and containers. In supporting a particular property type, the supporting class effectively just needs to produce a valid property editor panel which the user can use to edit the property. These panels are added to the property editor window automatically as shown above.

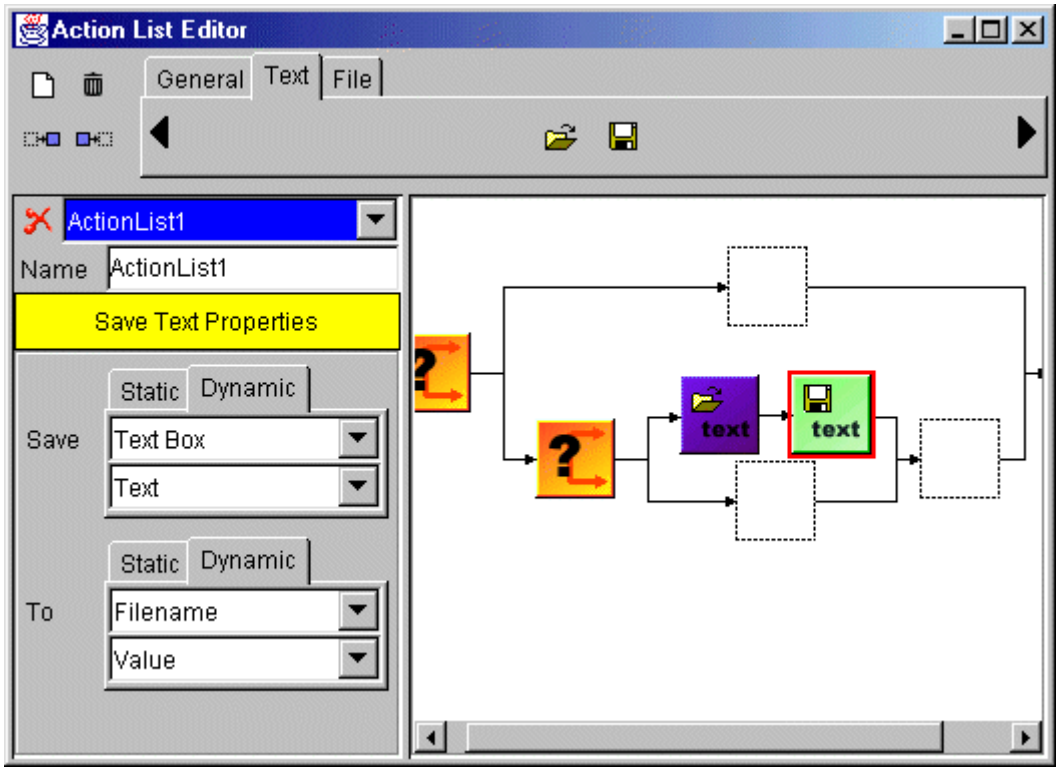
5.3: Beyond PIY

The structure of PIY-II described so far doesn't offer any more power to the user than the original PIY did. Where PIY-II goes further is in allowing the user to specify what happens on particular application events. The user can decide what they want to happen when a particular button is clicked, for example.

(Aside: New event types can be added to PIY-II by extending the PIYEventListener and placing the new event listener into the "piy/eventlistener" directory.)

5.4: The Action List Editor

An *action list* is just a sequence of *actions* that are executed sequentially. An *action* is a single specific function. When the user wants to specify the consequences of pressing a particular button, they do it via an action list. The action list editor window is shown below.



The action list editor enables the user to easily create and edit action lists. The actions which are available to users to add to their action lists are provided in the "piy/action" directory - to add new actions to PIY-II it is a simple matter of placing them in that directory. Once PIY-II has detected all of the actions (on startup), it places them on the toolbar in much the same way it handles user components and containers. Also like user components and containers, actions have properties which the user can edit.

There is one difference between the properties of an action and the properties of a button, say, and that is that the properties of an action can reference properties on other components. So whereas a window may have a background property which is a fixed colour, an action could reference the colour of another component - that is, properties of actions can be dynamically bound to the properties of components.

As an example of this, there is a "Change Property" action. This action can be used to change the value of a particular property to be something else. So it might change the text in textbox1 to the text in textbox2, for example. All types of runtime effects are possible by the use of this action alone.

5.5: The Memory Editor

The memory editor is the last of the main windows which make up the PIY-II user interface. The concept of a memory in PIY-II was not something present in the initial user stories. The idea came about whilst implementing a few new actions that could benefit from having some form of intermediate store for values, rather than relying on storing values in component properties. So the memory simply consists of a set of variables which have specific types and values. Users can use the memory editor (right) to create, edit and delete variables for use in their PIY-II projects.

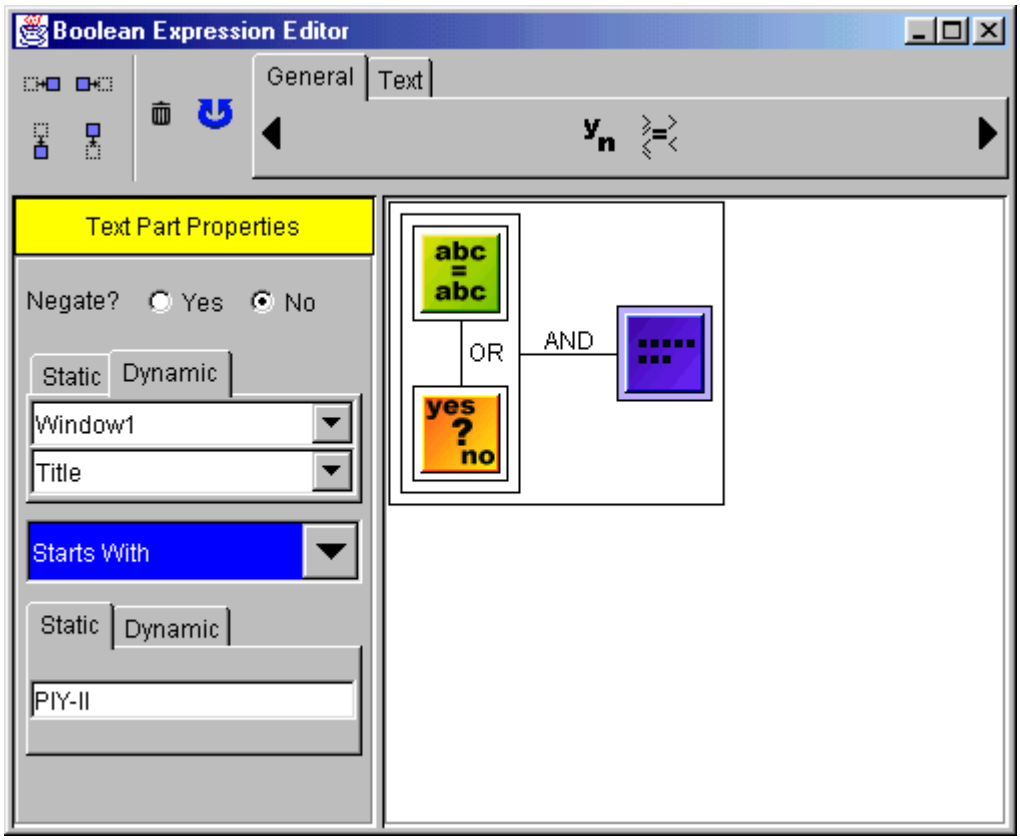
The data types supported by the memory editor is precisely the set of property types that is supported. That is, each type for which there is a property type editor available for will be recognized as a type for use as a variable.

5.6: Boolean Expression Editor

Since there is an *if* action, there needs to be a way for the user to express boolean expressions. The method by which PIY-II handles this is to have a `BooleanComparable` class which expresses something that can be evaluated to give a boolean answer. It can also be specified as to whether the answer should be negated. There are then `PIYComparator` classes which extend the `BooleanComparable` class, and are handled via a plug-in mechanism. Each new `PIYComparator` that a vendor might provide has two variables and a few methods of comparison. For example, it might specify that it can compare two strings, and that it can compare them in two ways, i.e. check whether they are equal, or check whether they are equal ignoring case. Each `PIYComparator` can have as many modes of operation as it likes, although it's normally best to have functionality spread out over a few different classes. All `PIYComparators` are loaded into PIY-II at startup using the same extension mechanism used for the other plug-in components. Also, PIY-II already has the mechanisms in place to deal

with offering support for the variables that the `PIYComparator` will work on - the variables are treated in the same way that properties are treated on other components. So most of the code used in the boolean expression editor is code that is reused.

The one other thing that is needed is a way to link together different `PIYComparators`. That is, a way to express (Comparator1 && Comparator2) || Comparator3, for example. This mechanism is provided for by the `BooleanExpression` class. This class extends the `BooleanComparable` class since it can be evaluated, and simply stores internally a pair of `BooleanComparable` classes. These are linked together either with an "and" operator or an "or" operator. This is all that is required to express all possible boolean expressions. The boolean expression editor is shown below.



5.7: Compiling

It has been noted that the main goal for PIY-II is that it should be possible to create and compile real applications. The XP system metaphor developed for PIY-II ensured that the thing being edited (a `Project`) was kept separate from the thing doing the editing (PIY-II). Because of this, compiling a project into an application became a simple matter of taking a particular project and setting up the relevant listeners onto each component so that the specific action lists are executed on particular events.

There are really two forms of compilation in PIY-II. The simplest is compiling an application for running within the PIY-II environment. This is done via the process just mentioned (ie. the component listeners are set up to fire the correct action list when the relevant event occurs). There is one minor complexity to doing this - namely in how PIY-II monitors the application being run. That is, when the application terminates, PIY-II needs to know about it. PIY-II

also needs to have a mechanism to terminate a running application. I managed to get around this problem and avoided using the deprecated `Thread.stop()` method. The mechanism put in place is a process made up of three elements:

1. An application has a `running` variable that is initially set to true. As soon as an application is required to terminate (by the application itself or some outside source), this variable is set to false.
2. There is an `ApplicationMonitor` class which monitors the value of the `running` variable in the running application. As soon as the variable becomes false, the monitor notifies a predefined observer.
3. The observer, once notified by the application monitor, takes the steps necessary to terminate the running application. The simplest way to terminate an application is via the `System.exit(0)` method, but if PIY-II is running at the time, a call to `System.exit(0)` will result in PIY-II being terminated as well. So the `Application` object has a `closeAllWindows()` method which essentially kills the application's AWT thread.

Applications that are compiled for use outside of the PIY-II environment require no further involvement of PIY-II. They are essentially pure Java applications - they will run on a Java virtual machine with or without PIY-II installed. The way this is achieved is by creating a *jar* file containing all of the PIY-II files necessary for the compiled application to run. These files are not really particular to PIY-II, except in being the base classes from which general actions/components/contaiers/etc. are built. Currently PIY-II saves all of the components, containers, event listeners and action classes that are in PIY-II. An optimisation of the process would be to store only those classes that are actually used by the project. That said, the *jar* file containing the PIY-II support files is currently only around 30KB in size - hardly something that needs optimisation at the moment.

5.8: Summary

The general operation of PIY-II is in linking action lists to particular application events. The extensible framework on which much of PIY-II is based means that the existing incarnation of PIY-II (containing relatively few actions and components) can be enhanced over time to include many new features.

PIY-II offers to the user something that no other currently available product does - the ability to create applications without writing code. *Java Studio* would be a possible rival were it still being produced, but having used it I can't say I found it particularly easy to use.

PIY-II lacks the sheer number of options available in a more mature piece of software, something I am glad about. The more mature a piece of software becomes, the more unnecessary features it accumulates. Since PIY-II is aimed at beginners, the point where the number of features offered overwhelms the users is the point where it will start to fail its purpose. The user interface as designed and implemented is, I believe, advanced enough to be useful but spartan enough so as not to be feared.

The next chapter explores, amongst other things, some of the problems I experienced whilst applying XP to the development of PIY-II.

Chapter 6: Where XP fails

When XP doesn't work, you're not doing XP.

The above is a true statement by somebody defending XP. There are few statements of people attacking XP which haven't been defended by the above statement though. Is XP really so good that a team following it to the letter will always produce a successful system?

6.1: Failure in Chrysler C3

The C3 project (Chrysler Comprehensive Compensation) was the project that effectively invented XP. Its aim was to replace the many payroll applications at DaimlerChysler with one single application. The project came about when Kent Beck was asked to performance tune Chrysler's payroll system - he told Chrysler there were deeper problems with the system and that they should start over from scratch. They agreed, and Kent then accepted the job of coaching the new team. Chrysler also agreed to allow Kent to use his new process, XP, to develop the system.

The project was started in around march 1996, and Kent Beck brought in Ron Jeffries as another team coach. On the 25th June 1999, Ron Jeffries ceased to be a part of the C3 project full time. On the 1st February 2000, the C3 project was terminated. The reason Kent Beck gives for the C3 project being terminated is:

> *"Near as I can tell the fundamental problem was the Gold Owner and Goal Donor weren't the same. The customer feeding stories to the team didn't care about the same things as the managers evaluating the team's performance. This is bad, and we know it's bad, but it was masked early because we happened to have a customer who was precisely aligned with the IT managers. The new customers who came on wanted tweaks to the existing system more than they wanted to turn off the next mainframe payroll system. IT management wanted to turn off the next mainframe payroll system. Game over. Or not, we'll see..."*

(Note: the "Gold Owner" funds the project, the "Goal Donor" defines what needs to be done. These may or may not be the same individual. In C3, they were different.)

So, due to a management change, the goals of the system changed. The previous customer wanted a new payroll system (as Kent had suggested), but the new customer wanted tweaks to the existing system. This reason for termination of the project seems outside of the scope for a methodology to deal with. However, it does highlight another potential problem - what would have happened if the new customer *had* wanted the new payroll system? What would have happened if the management wasn't changed as dramatically as it was? It is likely that many changes in the requirements of the payroll system would result, but how effective would the communication of the *current* state of requirements be?

There is, generally, only one on-site customer. This customer should ideally represent the views of the management and the users of the system, and is responsible for clarifying the meaning of particular stories to the programmers, as well as having the authority to change or

modify requirements at any time. In the case of a minor management change, what happens if this employee is relocated?

6.2: The Customer's Employees have Differing Goals

The team specifying the requirements must have a common goal. A mechanism to ensure that all of the owners of requirements have a common goal is something missing from many methodologies though. Some heavyweight methodologies have their requirements set in quite immutable states to hide the fact that they can change - changing a requirement requires a lot of effort on the part of the person wishing to change it. Changing a requirement in a lightweight methodology requires very little effort - that's the whole point. When a requirement change is needed, the customer can eliminate the old requirement and replace it with the new, communicating the change to the programmers. There is no monitoring of these changes though. Whereas in a heavyweight methodology a changed requirement is done out of absolute need (in theory), a requirement can be changed in XP on a whim. This is a strength of the lightweight processes, but if not handled correctly it is potentially dangerous.

6.2.1: Communicating Requirements Changes to the *Customer*

So the problem seems to be in communicating requirements changes to the customer, not in communicating the changes to the programmers. The programmers are working on the system all day every day, the customer will only have a few employees with knowledge of the requirements of the system - and only one of these can actually make changes. Whilst the on-site customer will know about all minor changes to the system requirements, the on-site customer's colleagues will only know of the initial requirements. The on-site customer needs a way of effectively communicating requirement changes to his colleagues - something which could be difficult due to them working in different offices.

Perhaps a project isn't in danger of the on-site customer leaving. But maybe the on-site customer will be ill. Maybe they'll need a week off work. Again the problem of not having an on-site customer who knows enough about the current state of the requirements creeps in. A possible solution to the problem is introduced in the next chapter.

6.3: Is XP too "High Discipline"?

Alistair Cockburn described XP as a *high discipline* methodology, meaning "one in which the people will actually fall away from the practices if they don't have some particular mechanism in place to keep them practicing". The mechanism in place at the C3 project was the coach, Ron Jeffries. He went on to say (prior to the termination of C3), "Ron is that mechanism at the moment. Should Ron leave, then unless he is replaced in his role, I quite expect to see the team not following the practices properly in less than 6 months". About six months after Ron left the C3 project, it was terminated. As has been mentioned above though, the reason for the termination was political not technical, but it is worth considering the role of the project coach in XP - the person responsible in making sure the team is following the process correctly.

The question here is not whether XP is too "high discipline", but as to whether there is a sufficient mechanism in place to ensure that the project team are following the practices effectively. More specifically, is appointing a team coach sufficient to ensure that the team

doesn't deviate from the core XP practices? Whilst at C3, Ron Jeffries remarked *"...we have recently found it necessary to have a revival meeting to recommit to our beliefs, because we had strayed fairly far from them. I'm quite concerned - the whole team is - about what let us get off track so far with no alarms going off."*

So another question arises - why does a project team stop following XP practices? Maybe the practices themselves are hard to follow, and shortcuts are sought when time is pressing. In particular, the XP practices which require the most discipline are pair programming, test first design, YAGNI, refactoring, and having all unit tests pass before integration.

The main problem is that XP requires a whole set of practices to be followed *all of the time*. This is what is claimed to make XP work - but it certainly requires a lot of discipline.

6.4: There's no clear direction

One objection often raised towards XP is that there's no architecture, which XP advocates respond to by pointing to the system metaphor. As I have stated before, the term "metaphor" has not been adequately described in XP literature - the most pure form of XP certainly doesn't consider the metaphor to be a solid artifact. Even as an initial design, the system metaphor falls far short of being a true architecture.

XP doesn't tell the programmers what to do, it just tells them how to do it. Slightly more heavyweight methodologies have a stricter set of guidelines to follow to get from A to B, as in "first perform this particular analysis technique, then produce this document, then call a meeting, ...", and so on. The programmers know exactly what is required of them. In XP, to get from A to B the programmers have a requirement at A, and need to have some fully working and tested code at B. Their only hope is that in following the XP principles they can get it right.

Of course, producing code is what programmers do. They'll program anything. This is part of the problem - if programmers are not given a strict enough route to follow they will begin to deviate from what is actually required. Pair programming helps, but if the programmers don't know where to start or aren't sure of the particular goal they are trying to achieve, no amount of quality in the code is going to make it any more *correct* (in the eyes of the customer, at least). The key seems to be *communicating* the requirements, and then providing some form of *guidance mechanism* to show the programmers where to start and what route to take. The core practices help with the quality of the code produced, but they don't tell the programmers where to actually start.

6.5: Where do functional tests come from?

The functional tests provide the only way to show that a system developed using XP practices is correct. That is, if the functional tests pass 100%, the system provides all of the functionality required of it by the customer. The customer still may not be happy with the system though - there could be missing tests. There's no standard way of producing or managing functional tests. There's no checks put into place to ensure that the functional tests are not contradictory to the user stories. It's only when a functional test fails that the programmers might realistically question it - such as "Why does this functional tests fail? I'm sure I've implemented the story as presented". The reason could be either that the story is

wrong, or that the functional test is wrong. Worrying about the correctness of a requirement is too late at this stage, ie. when the code has already been written.

There needs to be some form of direct mapping between the user stories and the functional tests. No story should be left without a test, and no test should be left without a story. The customer may still not be happy, but it's important that the requirements they specify are precisely the same as what is being tested.

6.6: XP doesn't scale

XP works best on small sized teams, maybe a dozen programmers. This is partly due to the point raised previously about there being no clear direction. Larger software projects with programmers spread around in locations miles apart needs to have solid organization to ensure everyone knows what is required of them. Face-to-face communication with the customer is the thing most valued in an XP environment - if some of the programmers are on the other side of the world this is simply impossible. In larger projects, the programmers really do need to know the bigger picture, and the lack of a requirements document and a definite design plan is certainly a major concern.

It could be argued that large programs can always be split into smaller components, and that each separated group of programmers could work on separate components. This *may* be feasible, but should the collective code ownership practice still be observed across *all* of the code? Without it, refactoring becomes a case of just refactoring a particular component. The overall design of the system doesn't change, only the design of its sub-components does. That is, there *must* be an overall design of the system based on some form of requirements analysis - this overall design will become virtually impossible to change over time. There are, perhaps, a few solutions to this, but XP currently offers no procedures to accommodate the management of large software projects.

6.7: General Observations

There are further concerns I have with XP, although they don't seem to be as important or fundamental as those listed above.

6.7.1: Unrealistic Practices

Is the forty hour week practice really required? If the programmer are forced to do overtime consistently for many weeks, it shows bad project management and poor time estimation. There should be more emphasis on sorting out planning and estimation, rather than having an explicit practice of the programming team constantly watching the clock. Further, is excessive overtime common in software development? Images of dedicated programmers holding all-night coding sessions fuelled by nothing but enthusiasm and caffeine certainly seems to be what the average person might picture "extreme programming" as, but I just don't believe it to be that common a phenomenon.

6.7.2: Dangerous Reliance on Verbal Communication

For a start, verbal communication doesn't scale well to larger projects. If the main method by which programmers clarify details about particular user stories is by asking the customer, it is

because the user story is not formally specified enough. That is, there is ambiguity in the requirements. As soon as the customer becomes unavailable, as may be the case on larger projects, such clarification cannot be obtained by the programmers. So the system develops features which are based on the programmers' assumptions, and it begins to deviate from what the customer will believe to be "correct".

Maybe the functional tests will catch these deviations. No real process is defined by XP in how to create functional tests though, and there's also the question as to why a functional test could specify the requirement without ambiguity when the story couldn't.

6.7.3: Repeat Refactorings

Collective code ownership means that one programmer can refactor the code of another. It also means a third programmer can come along and refactor the already refactored code. As can a fourth programmer. There aren't any guidelines in refactoring (nor can I think of any myself) that limit the amount of time sections of code are refactored for. It's plausible that a lot of time can be wasted by well meaning programmers going round in circles refactoring each others code. *Real* code has to be written at some point.

I doubt this problem is particularly common, but it could become slightly more of an issue when refactoring tools become more powerful. A lot of time is spent by average computer users "tweaking" their desktop, changing the colours of window title bars, changing the background wallpaper etc. With more powerful refactoring tools, programmers may develop the tendency to spend unproductive time applying refactorings whilst *appearing* to be doing something productive.

6.7.4: Spartan User Interfaces aren't Useful User Interfaces

One (non-explicit) practice in XP is that developers produce spartan user interfaces for use in their system whilst it is still being developed. The idea is that user interfaces are easy to create, so as soon as the customer knows more precisely what they want it is cheap enough to quickly build a more advanced GUI to accommodate these needs. Maybe in some situations this is true: *in most it is blatantly untrue*. Quality user interfaces are **not** easy to create. There certainly exist good tools to aid their creation, but good tools are developed for many difficult tasks. Some of the best tools are developed for the most difficult tasks.

One requirement that customers never know how to write correctly is "the system must be easy to use". It is the one requirement that developers seldom pay much attention to because it's much more difficult to accomplish than it sounds.

Also, I've found that in developing PIY-II, the user interface is also one of the most difficult things to test in an automated way. YAGNI says you shouldn't implement things that aren't specified, or at least aren't specified for construction right now. However, I would argue that a spartan user interface is not the one that will be used in the final product. Creating one now means throwing it out later - less time can be wasted if it is invested in creating the user interface that will actually be used. Computer users are often much more impressed by a nice user interface than they are by range of features or stability - at least initially. Compare Windows and UNIX, for example.

Chapter 7: Extensions to XP

Extreme Programming is not a silver bullet

Fred Brooks wrote a paper entitled *No Silver Bullet — Essence and Accident in Software Engineering* (analysed by Brooks in the 20th anniversary edition of *Mythical Man Month* [BROOKS95]), in which he defines the term *Silver Bullet* as:

> *… [a] single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity [of software projects].*

XP doesn't, in my opinion, offer an order of magnitude improvement over other methodologies - it is not a silver bullet. A discussion of this is carried on [XPMAP00], but the important point to remember is that *any* improvement is better than no improvement at all.

No methodology is easy. Skill is needed to succeed. There are, however, areas in which the programmer's skills can be more effectively targeted once a general procedure is introduced. The unit tests are a classic example of this. At first, they seem like a difficult concept to actually implement, until it is realised that most of the work has already been done - there are unit testing frameworks for virtually every programming language you're likely to use XP in. The programmers skill is immediately targeted to writing the tests, rather than to wondering how to manage or design them.

However, there are some areas in XP where we're only pointed in what direction to go - there's no map. In these extensions to XP, I offer a few guidelines as to how to steer to each target. Skill is, of course, a prerequisite, but these extensions show what has worked for me, and highlight some of the difficulties an XP programmer may have to overcome in trying to follow XP as purely as they can.

7.1: From User Stories to Functional Tests

It is up to the customer to generate both the user stories and the functional tests (with a developer-side programmer). Neither of these tasks are trivial, and getting either of them wrong slows down the development process. Every story must have tests, every test must relate directly to a story.

It is a goal of software engineering methodologies to match the users' expectations with those of the programmers. The first step in realising this comes in ensuring the user has specified the requirements they really mean.

7.1.1: User Stories

The user stories are a collection of requirements for the system to be developed. Each one should be free of technical information and only go into enough detail that a low risk estimate can be made on the time it will take to implement the story.

7.1.2: Functional Tests

A functional test should be present for every functional requirement in every story. Kent Beck [BECK99] writes:

> *The question they [the customers] need to ask themselves is, "What would have to be checked before I would be confident that this story is done?" Each scenario they come up with turns into a test, in this case a functional test.*

7.1.3: Mapping User Stories to Functional Tests

So, according to Beck, it should be possible for the user to apply a consistent process to each of their stories to obtain the relevant functional tests. By applying such a process to *every* story, a complete set of functional tests can be constructed. If the user feels that a particular test seems to be missing, this will more often than not indicate a missing story. So a process for generating functional tests is a valuable feedback mechanism for highlighting missing stories.

Such a process was developed and used to facilitate the construction of functional tests for PIY-II.

7.1.4: Example functional tests for PIY-II

Given the following story from PIY-II,

> *2. Only one project can be loaded into PIY-II at any one time;*

the functional tests needed are quite minimal. The functional test I created for this was:

Ref	Test	Expected Outcome
2.1	Load a project with a project currently open.	Either a) the new project is opened, and the current one is closed; or b) the current project remains open.

Here, "Ref" represents the reference number for the test, "2.1" signifying it's the first test for story number 2. The expected outcomes should signify that the story as given has been correctly implemented (i.e. the story doesn't specify how the current project should be closed). A more complicated story and testing structure is given below.

Consider the story

> *3. If the user attempts to load a project when an unsaved project is currently open, the user should be presented with the options a) save the current project; b) load the new project anyway; or c) Cancel - continue editing the current project.*

The functional tests for this story are as follows:

Ref	Test	Expected Outcome
3.1	Load valid project with an unsaved project open	Dialog should appear with the option to save the current project, or load the new project anyway, or to cancel the load operation.
3.1.1	Perform tests 4.1, 4.2	Project should be loaded
3.1.2	Don't save the unsaved project	Project should be loaded
3.1.3	Cancel the load operation	Project should not be loaded, current project remains.
3.2	Load a project with a saved project currently open	Project should be loaded.
4.1	Save the current project under a different filename that already exists	Dialog should appear with the options a) Overwrite the file; b) Choose a different filename; c) Cancel the save operation.
4.1.1	Choose overwrite	Project saved under different filename.
4.1.1.1	Reload project that was just saved.	Project loads and is identical to the one just saved.
4.1.2.1	Choose a different filename that exists	As for test 4.1
4.1.2.2	Choose a different filename that doesn't exist	Project should be saved
4.1.3	Cancel the save operation	Project should not be saved - the file with the filename that was in use should remain untouched.
4.2	Save the project under a different filename that doesn't exist.	Project should be saved.

The way the above tests are intended to be run are as follows:

- A test with the reference "N.x" is the x^{th} test for story number N, and can be run at any time. *Note: N.2 doesn't need to be run after N.1, see next point.*
- A test with reference "N.x.y" is a test that requires "N.x" to be run immediately before it - this is generally because the previous test will put PIY-II into a particular state that the test "N.x.y" will be testing. Hence test 4.1.1 can only be run immediately after 4.1.
- Similarly for tests with references "N.x.y.z" etc.

Note, story number 4 is given as "4. If the user attempts to save the current project with a filename that is currently in use, the user has the options a) Overwrite the file; b) Choose a different filename; or c) Cancel the save operation - i.e. don't save."

7.2: From User Stories to Metaphor

It is unclear in XP exactly how a developer should go about generating a system metaphor. In PIY-II I have taken the view that it should be possible to generate an initial system metaphor

directly from the user stories. In fact, if the system metaphor *doesn't* come from the user stories, how can it be shown to be correct?

Consider the following definitions:

1. The user stories are a set of functional requirements which the final system should implement.
2. The system metaphor is a collection of classes and patterns that make up the core of the system (some might call this the 'architecture').

These two concepts are completely different - but should it be possible to move from one to other without loss of meaning? The answer is *no* - not all of the functional requirements as presented in the user stories will be core to the system. It is however the case that the user stories that *do* make up the core of the functional requirements in the user stories should be correctly represented in the system metaphor. So a technique to generate the system metaphor from the user stories must take into account *which* user stories are core to the functionality of the system.

It must first be said that there are a number of ways to generate a system design from a set of requirements. However, these generally take a lot of time in analysing the system and generating paperwork. The XP metaphor should be a small design taking into account only the core functionality of the system. Further, it doesn't need to be completely finished - refactoring of code whilst implementing the metaphor and associated stories often means the final system metaphor can be quite different from the initial one. This is in keeping with the XP philosophy of not making final decisions about the system when you're still unsure about what the system is. So the technique that I used to generate my system metaphor was designed specifically so that it was quick and gave me scope to make changes to the metaphor later. The metaphor must be amenable to future adaptations rather than being predictive of how the system may change.

The technique I used is thus described:

7.2.1: Eliminating user stories

Some user stories can immediately be discarded as not being part of the core functionality required in the system. (Some example user stories from the PIY-II project are listed in appendix C). Consider the stories highlighted as "constraints". These stories put constraints on what some of the final classes should be able to do, but don't define any of the abstract attributes of them. For example, story number 31 says "Components must include buttons, radio buttons, combo-boxes and lists". So it identifies that buttons, radio buttons, combo-boxes and lists are all types of "Components", but specifies no abstract attributes of any of the given components. So the idea of "Component" seems quite central to the system, the given subclasses just being classes that behave like "Component". As another example of why the constraint stories can be disregarded at this stage, consider stories 34 and 35. These define that there must be the actions "if...then...else" and "calculation". So these things that have been specified behave like some abstract entity known as an "action".

So in general, stories which don't seem to define any abstract behaviour probably won't feature in the system metaphor, because it would not be expected that non-abstract classes

would be built upon (extended). The system metaphor should ideally be a framework on which the rest of the system can be built.

One point about the metaphor though - it doesn't contain only abstract classes. Why not? Well, some stories will reference other stories; the stories being referenced could define abstract behaviour, but the referencing story does not. Both stories might represent classes in the system metaphor, i.e.. one concrete class referencing one or many non-concrete classes. This construction is typical when some kind of collection structure is implied. An example of this happening in the PIY-II stories is story number 6, which says "There must be a GUI editor in which the user can add/remove or modify components on the user windows". Here, the "user window" seems to be concrete, but it has "components" on it, which seem to be abstract entities.

7.2.2: Selecting initial core user stories

Once you've eliminated stories which don't specify abstract properties or reference other user stories, you should be left with stories that imply abstract classes and stories which imply collection structures. There may be some stories that don't fall under these categories - for example, stories which determine the behaviour of some of the classes, but which are not classes in their own right. These types of stories can often be attached to the system metaphor using design patterns.

One point worth making before continuing is that some of the classes that will end up in the system metaphor will not be explicitly named or even mentioned in the user story. It is up to your own ingenuity to identify common behaviour in some of the stories and abstract out the key concepts. Also, some of the classes that end up in the system metaphor could come from the *glossary*. Since user stories should only specify testable functional requirements of the system, the glossary is constructed to contain descriptions of terms used in the stories.

So classes in the system metaphor can come from user story descriptions, the glossary, and from identifying common characteristics in the stories.

7.2.3: Where to start

The first question to ask yourself is "which class should I start with?" At least two approaches are possible.

1. Create a class which represents the name of the system - i.e. in PIY-II's case, start with a class called "PIY". Many of the user stories are of the form "The system must contain ...", so associations can be built up from the starting class "PIY".
2. Another approach is to start with a class which represents what is being manipulated in the system. This could be a "Document", a "File", or in PIY-II's case, a "Project". In this case we have "Project" in the glossary, so we already have a description of what it is.

The second approach has implications to the design of the system - what if the thing being manipulated in the system is a database? In this case, having a starting class of "Database" might not be the best idea. You generally connect to a database and modify it through the connection, rather than storing an individual copy of it in a system object.

Assuming you're not connecting to a database though, the main thing being manipulated by the system will generally need to be loaded/saved. This means that you need to make sure that the state of this class can be saved and reloaded, without the loss of important data. In Java this is relatively easy, by making the class (and classes referenced by it) Serializable. Using other languages you might need to incorporate the Momento pattern (see GAMMA94). This of course has implications for the design of the system metaphor, so it's important to know which programming language the system will be developed in.

So you could, essentially, design two UML diagrams representing the system metaphor.

1. A UML diagram with the system name as the main class.
2. A UML diagram with the entity that is manipulated in the system as the main class.

(The first diagram may contain classes that are present in the second diagram).

7.2.4: Attaching to the main class

Now there are two diagrams with only one class each. Start by developing the second diagram first - as some of the classes in this might be needed in the first diagram.

If we assume that the main class here is "Project", the first thing to do is to locate any stories which reference "Project" directly. In PIY-II's case, Project is defined in the glossary. It is defined as "a user application still in development. It consists of both the GUI Layout (in individual user windows) and the logic behind the components in the GUI." So we can imagine GuiLayout as a class, along with UserWindow and Logic. Here GuiLayout represents where each component is placed, the UserWindow stores the components, and Logic represents how the component operates. As all of these classes just thought of seem to have something to do with "component", we'll create a "UserComponent" class (since we know that Java already has a class called Component, and it doesn't act like the component we're specifying here).

We now have a nice collection of classes to play with, but how many of them are abstract? That is, do any stories specify any likely subclasses of the classes we have identified? We know the user stories mention subclasses for UserComponent, but what about GuiLayout, UserWindow and Logic?

In fact, the user stories don't seem to mention anything that sounds similar to GuiLayout at all. In fact, from experience with Java we know that "Component" specifies x and y coordinates and width and height sizes itself - if we model that behaviour in our UserComponent the need for a separate GuiLayout class seems redundant. Assuming we weren't using our experience from Java though, it seems quite nice at this stage to have something that manages positioning of components on each individual UserWindow.

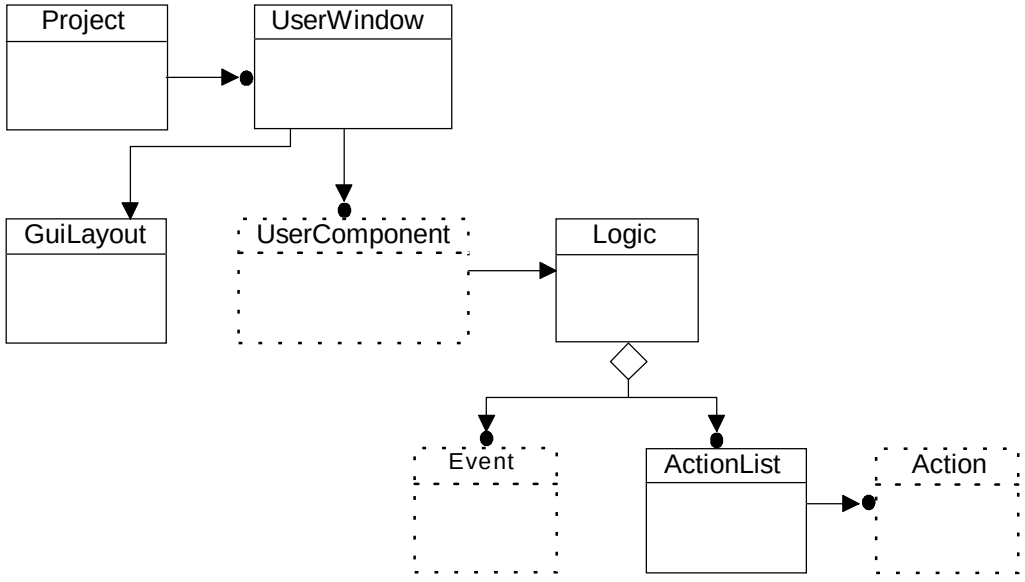
So what about the Logic class? This is a word not defined in the glossary, and not used in the user stories. Of course, as an abstract concept it might serve well as a class, but is "logic" in this context synonymous with "action list"? In XP we are allowed to ask the customer what they mean. Does the "logic behind the component" refer to the action lists attached to the events of the component? For the purpose of example we'll assume the logic behind the component to actually mean the *set* of event - action list pairs attached to a component.

So, what have we got so far?

1. Project
2. GuiLayout
3. UserComponent
4. Logic
5. UserWindow

We know that "logic" represents a set of event - action list pairs, so maybe we should also have an Event and an ActionList class. Also, since ActionList is a list of actions, maybe we should have an Action object as well.
We know there are different types of Event, so Event could be an abstract class. We also know there are types of actions, so Action could be an object. What about ActionList? Well, that is a list of Action objects (which are abstract) which is referenced from a particular Event (which is also abstract). So these classes we have just thought of are either abstract or reference a collection of abstract classes. So we can add the classes Event, ActionList and Action to our initial list.

These seem to represent everything that a Project is made up of, and the stories don't give any other requirements that would break the above classes down further. So we can attempt a UML diagram of these classes:



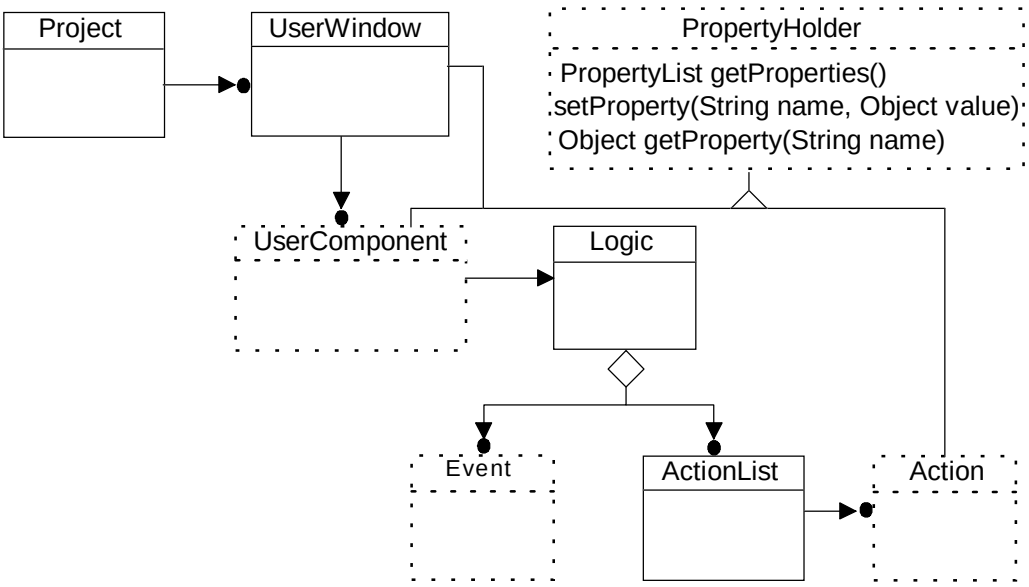
7.2.5: Check the Initial Sketch

We can now check whether this original design makes sense with regard to the stories. To do this, go through each user story that mentions something on the above diagram, and see how easy it will be to implement the story.

The first problem seems to come with story number 7, which says "The GUI editor window must contain a section to edit the properties of the GUI component, and a section to edit the actions attached to the events associated with the component". The question is, how do we get the properties of the component? More to the point, since our "UserComponent" class is abstract, we don't even know at this stage what the properties of the implementing classes will be. Thus we have identified an abstract concept in the stories - the idea of "properties". We need to implement a method of accessing the properties of all possible implementing classes of UserComponent. We will also need to be able to access and modify the properties of UserWindow (size, colour, etc.), and also the properties of Actions.

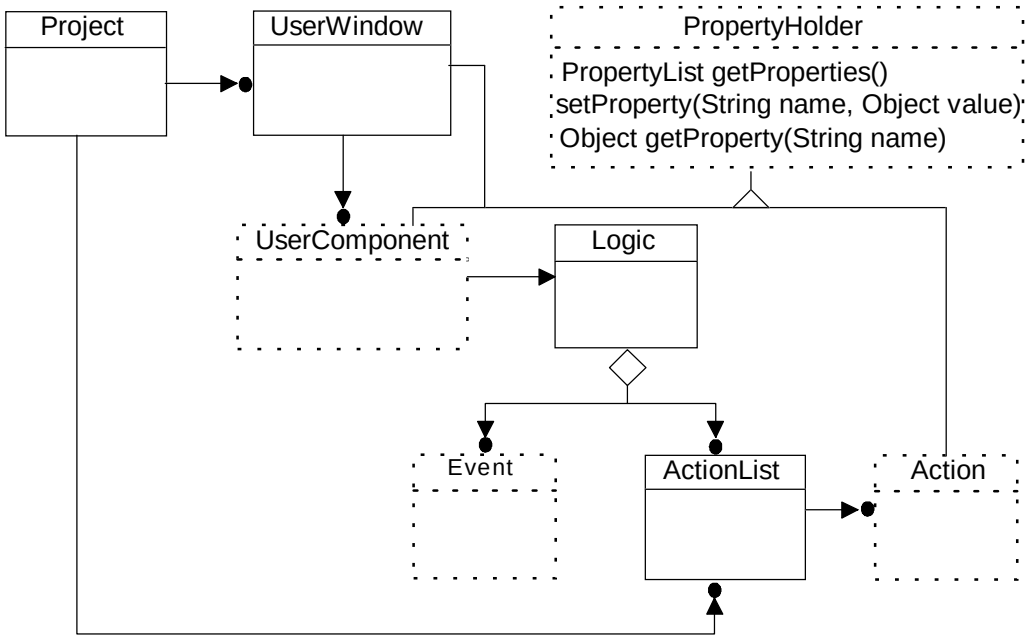
We also have the problem of where the properties of the UserComponents are currently stored - i.e.. the position details and width and height are stored over in GuiLayout. If we ask a particular UserComponent for its properties it will need to contact the relevant GuiLayout object to discover its position, width and height. In our class diagram, we only specify that GuiLayout has a reference to the UserComponent, not the other way round. More to the point, why is a separate class storing properties that are the business of UserComponent? To keep our classes properly encapsulated, it makes sense to move the properties of the UserComponents inside the UserComponent class.

Here is a changed attempt at the UML diagram in light of the above observations:



Now it should be easy to get/change the properties of the UserWindow/UserComponent and Actions. We have also managed to get rid of GuiLayout, simplifying the design and keeping the properties of UserComponent within the class.

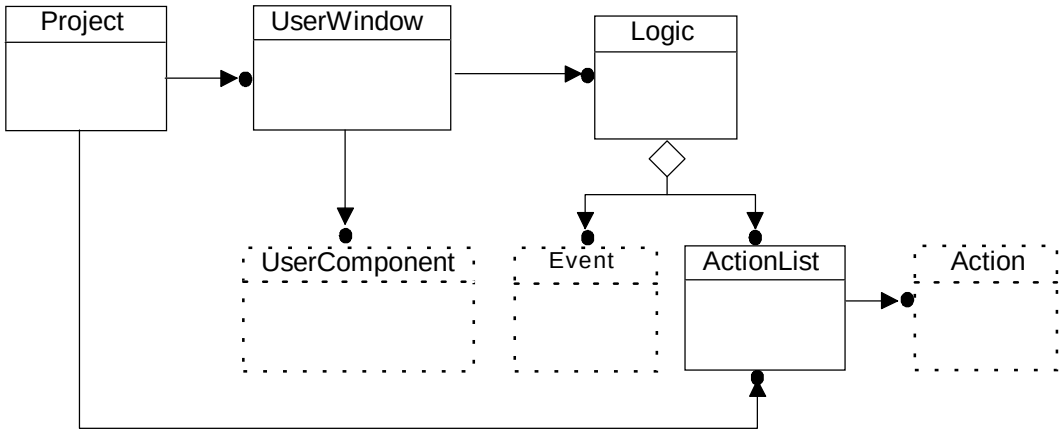
We only have one more story left to capture for the above to be complete - story number 19. It says "Users can create an action list that is not attached to any events." So there will exist, in any project, a collection of ActionLists which are not directly attached to any component events. This requirement can easily be captured by story ActionList references in the Project class, and is shown in the following diagram.



The other UML diagram can be generated in a similar way.

7.2.6: Implementing the Metaphor

Whilst the system metaphor has now been fully implemented for PIY-II, it is not the same as the one described in the previous section. This is because of what I stated before - the initial system metaphor as drawn on a piece of paper is liable to change once the implementation phase is underway. There's no point trying to force the implementation to fit the design if it becomes unnatural to do so. There would be serious implications for any project that cannot change its design at this early stage - what chance would it have against later requirements changes? The reasons for the changes will follow the UML diagram of the currently implemented PIY-II system metaphor:



The first major omission is the PropertyHolder class. This major change was made because of the necessity of UserComponent being an interface, and not an abstract class. The reason for this is the lack of multiple inheritance in Java - a class that wishes to extend the standard Java

Button class couldn't also extend the UserComponent class, for example. Because of this, the work in implementing the PropertyHolder methods falls to the actual implementing UserComponent classes. It was quickly decided that the amount of code duplication that would take place due to having to implement the PropertyHolder methods in every UserComponent was excessive, so the interface was removed. Of course, this raised the question of how I should get the properties of a UserComponent, a UserWindow, and an Action.

Another consequence of UserComponent being an interface is that I've moved the Logic class reference to the UserWindow class. This was again due to code duplication - if the reference was stored in the UserComponent, it would be up to the implementing classes to provide the code for it. Since I'm expecting to have many implementing UserComponent classes, the overhead of having to handle this extra responsibility is too great. The UserWindow thus now stores a reference to Event, ActionList pairs, where each event is a particular UserComponent event.

7.2.7: Replacing the PropertyHolder Interface

In removing the PropertyHolder interface I brought in measures which would allow properties to be retrieved and changed within components without them needing to explicitly declare them. The approach is similar in style to the way Java Beans work, using low level reflection to introspect the relevant classes for valid properties.

Any properties the UserWindow, UserComponent or Action wants to expose to the outside world it provides "get" and "set" for. For example, a simple button implementation has the following source code:

```
package piy.usercomponent;

import javax.swing.JButton;

/**
* Standard user button object, with edittable caption.
* @author David Vivash
* @version 1.0, 15/11/00
*/
public class UserButton extends JComponent implements UserComponent
{
      public UserButton() { super('Button'); }

      public String getText() { return super.getText(); }
      public void setText(String text) { super.setText(text); }
}
```

So this simple UserButton has only one specified property - "text", which has the type String. (Note: all UserComponent's implicitly have position and size properties automatically handled by PIY-II).

PIY will recognise properties as long as the following criteria are met:

1. There is a "get" method for the property, which takes no arguments.
2. There is a "set" method for the property, which returns nothing, but takes one argument of the same return type as the "get" method

3. The type of object returned by the "get" method must be supported by PIY-II.

So property types need to be explicitly supported by PIY-II. This is handled in a "pluggable" manner - new property type support can be added to PIY whenever needed, by adding a "PropertyEditor" class. Each PropertyEditor provides support for one type, providing an interface between the user and the property being edited. For example, a PropertyEditor for a String type might simply provide a text field which the user can change. A PropertyEditor for a Color type might display a colour selector dialog to the user.

7.2.8: Summary of the Metaphor Construction Process

The metaphor construction process just described can be summarised as follows:

1. Eliminate user stories (such as constraints).
2. Select core user stories which encapsulate abstract properties of the system, or describe collections of those abstract properties.
3. Construct a UML diagram starting with a class for the system, or with a class representing what is being edited by the system.
4. Check that the structure of the diagram makes sense with respect to all of the stories, and adapt it if necessary.
5. Commit the design to code.
6. Refactor the design.

This process gives a definite place to start with the system. It is perhaps worth remarking that the customer may have different views on what makes up the core of the system - this could be significant.

Note item number 6 in the list: *Refactoring*. The specific refactoring I performed was in removing the PropertyHolder interface. This would a *major* refactoring if done in code. Designing quickly on paper, however, meant that describing the process in the above text was a lot more difficult than actually modifying the design. By refusing the advantages that paper designs can have, XP is depriving itself of a valuable technique. Design documents in notations such as UML offer a powerful medium for modifying complex designs, and are often simpler methods of communication for those experienced in interpreting them. Whilst the refactorings eventually have to be committed to code, a visual map of what the code will look like after the refactoring is a valuable document for the programmers to have should they lose their way.

7.3: Customer Pairs

One problem highlighted with the idea of an on-site customer was that it didn't go far enough. The on-site customer can change requirements without his colleagues knowing about it. As soon as he needs to be replaced for any reason (even for a short while), the new (possibly temporary) on-site customer does not know enough about the true set of requirements for the system.

A possible solution is to introduce customer pairs - two customers working on-site. The office layout would be similar to the current situation where the customer has an office in the corner of the programmers work area. Any query a programmer has with a requirement can

be dealt with by either of the on-site customers, and since both customers are effectively sitting side by side, more underlying problems have a better chance of being dealt with.

The benefits of two on-site customers would be many - not least the fact that the system will be developed in the direction of the common goal, rather than in the direction of goal of one customer. Since requirements don't change dramatically (in general), it should be possible to change one of the on-site customers from time to time. This should enable the minor requirements changes that have been made to become more widely known to the customer.

Whether or not this practice could really be followed is another thing entirely. Getting a single on-site customer is difficult enough - getting more would only be possible when the customer is confident enough as to the effectiveness of XP. It is unlikely that small companies will be able to accommodate this practice.

7.4: Test the tests: Mutation Testing

Tests cannot prove the absence of bugs. So why not see if the tests provide adequate code coverage? The programmers on an XP team will be generating many test cases, but the possibility of *missing* tests is always there. There are plenty of test coverage tools available, and plenty of techniques for analysing how thorough a test set is. Since there's no real test strategy in an XP project, use of a test coverage tool could provide an extra level of code quality.

On particular method of testing a test set is known as *mutation testing*. The idea of such testing is that the original code is changed in some way, ie. mutated, and the test set is then run against this mutated code. If a test still passes the mutated code as being correct, then it's probable that the test is not adequately testing the code. It's possible that some mutated code will not be detected as being incorrect if there is simply no test case covering it. Typical mutations might be to change guard conditions in loops (ie. change '<' to '<='), or change if statements to their logical inverses.

I have mentioned mutation testing for the simple reason that a framework has already been written to automate the process in Java. *Jester* is freely available and works alongside the set of JUnit tests that have been developed. In using Jester, it's possible to analyse the coverage of a set of unit tests over a particular set of classes.

The unit testing that proponents of XP advocate is generally very thorough, but a practice of continually checking that the set of unit tests adequately cover the code could take XP testing to the next level.

7.5: Integrated XP environment

Methodologies often encourage the invention of new software development tools. There are tools to handle UML diagrams, tools to handle source code control, tools to design GUIs, tools to test coverage of tests, and many more. As a possible XP specific tool, I present the *Integrated XP Environment.* Why? Because XP is hard to stick to. The idea behind this little thought experiment is to look at ways in which each of the XP practices could be enforced, or at least accommodated.

- *YAGNI*

  This practice could be enforced directly. At the end of each iteration, say, a script could be run automatically that traces through the source code to find any methods which are never reached. These methods could be automatically deleted. Programmers might fear this system at first - they may fear that much of their work could be deleted. However, YAGNI means that methods should only be added *now* if they're actually going to be used *now*. A lot of getter and setter methods are often present in code regardless of whether they're ever used, and there's often many constructors for the same class when only one is used. Both getters/setters and constructors are generally easy to write - if some get deleted that were actually needed it's no great loss. Programmers may fear that more complex code will be deleted - but in this case they should ask themselves why complex code was written but never used. I wouldn't mind trying to run a YAGNI enforcing script against PIY-II to see how much excess code there is. I suspect there may be some superfluous setter/getter methods, but I can't think of any complex methods that aren't used.

- *Test-first Design*

  This could either be enforced or accommodated. The way I would imagine such a system working is that when a programmer is working on a class, they are provided with options to add/edit methods, or add/edit class variables, and so on. Whenever they click to add a new method, they are automatically taken to a window where they must write the test for the method first.

- *Unit Tests*

  If the idea of enforcing test-first design is deemed too intrusive, there could just be a simple option for the programmer to flip between code/test at any time. So if the programmer is working on a method, they could just press a key to be instantly taken to the relevant test. If a test doesn't yet exist, a skeleton version could be automatically generated by the environment. This approach would make it easier to work on both the code and the tests at the same time, and programmers would become more likely to follow a rigorous testing strategy.

- *Functional Tests*

  There's no framework for functional tests equivalent to those available for unit tests. Functional tests could be managed within the environment in a number of ways. The integrated XP environment should have a button that can be clicked to run the functional tests at any time.

- *Continuous Integration*

  The environment would have a button to run the unit tests at any time, so every time the unit tests are run and pass 100%, the system could instantly reintegrate the code back into the system. This would enforce the continuous integration practice to a degree, but the environment could be even more ruthless. If code hasn't been reintegrated for longer than a particular amount of time, the environment could

automatically run the unit tests to see if they pass. If they do, the code is integrated. If it doesn't, the programmers are warned that they're attempting to do too much in one go. If they don't try and get all of the tests to pass within a particular time-out period, their changes could be removed. This would encourage programmers to take smaller steps, and would also help enforce DTSTTCPW, YAGNI, and simple design in general.

- *Forty Hour Week*

  This could either be enforced directly, or via a warning system. The warning system offers a certain amount of research potential, and could also be used to determine areas of code more likely to possess errors. That is, the code written by any programmer who has worked for more than forty hour in the past week could be marked in some way. Whenever bugs start occurring, any marked code in the same area as the bugs would be the code more likely to be causing the errors. This would enable the programmers to automatically track down possible code hot spots a lot more quickly. There's also the research potential - what is the proportion of bugs occurring in marked and unmarked code?

- *Refactoring*

  It has already been mentioned that refactoring browsers already exist. It goes without saying that an integrated XP environment would be a refactoring browser.

- *Coding Standards*

  This is another non-XP specific practice, and it already accommodated to a degree. Keyword highlighting, parenthesis checking and other features don't help directly with this practice, but reformatting options could mean that all methods are automatically reformatted to a predefined standard, for example.

Chapter 8: Assessment and Future Work

PIY-II works.

With respect to actually getting software working, XP has been a success on this project - PIY-II works. More than that though, PIY-II is adaptable. Although with XP I have avoided looking too far forward, PIY-II looks far enough forward to be adaptable to future change. This is not a conjecture, it is a fact. Very late in the PIY-II project I decided that a boolean expression editor would be a nice addition. I managed to add such an editor in less than a single day. I had not considered at all in advance how such a thing would work, but the solution I have provided adds very few new classes to PIY-II. In fact, very few of the existing classes needed to be changed - those modifications that were needed where minimal to the extent of being practically unnoticeable. XP did not predict this development, but PIY-II was able to adapt to accommodate it, with a very high degree of code reuse.

8.1: Future work on XP

I have looked at the various XP practices, but there's still very little experimental research in the area. Pair programming stands alone in being the only practice which has real supporting evidence (see WILLIAMS00).

Laurie Williams, responsible for research on pair programming, wrote

> *The experiment I did on pair-programming could be done and respected as an academic study because it looked at the interaction of two programmers … But, I think studies to compare XP design practices vs. "traditional" design practices really can't be done in academia.*

That is, many of the XP practices are not really suited to academic research. However, here at the University of Sheffield, there runs two courses which can (and have) been used to test XP practices. One of the courses is a second year course known as *The Software Hut*. In this course, students are required to produce real software for real clients. With only a few clients, student teams are in competition with each other to produce better systems - the clients are responsible for marking how good they judge the software created for them. When I did this course my team used a more traditional software development approach. Our finished product was unfinished.

The other of the courses is a fourth year course, in which the students run a small software development company, *Genesys Solutions*. Again, the software they produce is produced for real clients. Both the *Software Hut* and *Genesys Solutions* offer environments not nearly as artificial as a pure academic one, and research has been carried out on both to judge the effectiveness of XP over the more traditional heavyweight methodologies. An initial paper introducing the experiments that have been carried out on XP in these environments has been produced, see [HOLCOMBE01-1].

The XP practices are generally interdependent - testing one practice in isolation is far too artificial an environment. So I believe future *academic* research would be best suited to finding ways of making the XP practices easier use and stick to. The previous chapter looking at possible extensions to XP introduced the notion of an integrated XP environment - one in

which the practices are explicitly enforced. Future work on XP could be in generating such a tool.

8.1.1: Research on Refactoring

Some tools have already been developed to automate particular refactorings, and there could perhaps be more research in this area. A refactoring must change the structure of code without changing its function, so a particular refactoring could be proved formally correct in this respect. A correctness proof would necessarily be programming language dependent - Java seems an ideal language for such proofs. My initial thoughts were that many implementations of mechanical refactorings would be simple to prove correct. I no longer believe this to be the case - even some very basic refactorings can have consequences not immediately obvious. Consider a "rename class" refactoring. It seems to be a simple find and replace refactoring - one that can be done without consideration to context. However, two classes in Java can have the same name provided they are in different packages. The *full names* of the classes include the package names, so are different, but the shortened versions are nearly always used. Java differentiates between the two classes by checking to see which packages have been imported. If both of the packages have been imported, the programmer needs to specify the full name; if only one has been imported, Java can infer that the class belongs to that package. That is - the actual class that is represented by a particular name is dependent on context - it depends on which packages have been imported. Simply replacing the old class name with the new one will not work - the old class name could reference the class in a different package. Renaming methods has similar problems (for example, if the two classes have a method with the same name, the name of the method in the second class shouldn't be changed).

As more complex refactorings emerge, proofs may be needed to an even greater extent - formally proving a refactoring, although difficult, is a lot simpler than general proofs of correctness for programs. Automatic refactoring, if reliable enough, could have future implications for the way software is designed - the meaning and function of code becomes the most important part, design would be secondary since powerful refactoring tools would make designs mutable at all stages.

8.1.2: Functional Test Management Tools

The integrated XP environment I outlined considered the idea that the environment should have test management facilities. I have also considered a simple way of recording the functional tests, based on the user stories. The next stage is to look at methods of actually generating the functional tests. Some work has already been done on this, see [HOLCOMBE01-2]. The approach given is via X-Machines, which may not be generally accessible to average software developers. The paper does assert that future work would include writing tools to support the method - such tools incorporated into the integrated XP environment would offer XP developers all of the foundations they need to complete a project whilst sticking to the XP practices.

8.2: Future work on PIY-II

Adding extra functionality to PIY-II in terms of new *actions*, *components*, *containers* and *event listeners* can be done simply - this is what the design has be structured to accommodate.

Real additions to PIY-II should be in the direction of improving usability and flexibility. The easiest way to improve usability will be:

8.2.1: Implement a help system

PIY-II's lacks any sort of help system - quite inexcusable for a system aimed at beginners. This should be the first new development of PIY-II. Enhancements could be made to the Descriptor class to provide general help for each new component added to PIY-II. There is already a simple help mechanism in the Descriptor class, (i.e. there's a method that can be used to retrieve the description of a component), but this is not currently put to any use from within PIY-II.

8.2.2: Create a "Wizard" system

The solution that many other applications use to solve complexity problems is to introduce *Wizards*. These generally guide the user through a particular process, describing each step as they go. The best way to implement such a system in PIY-II would be as a general *Wizard Plug-in Mechanism*.

The way I imagine such a system working is that there would be a general *Wizard Manager* in PIY-II from which the user could select a particular wizard, view its definition, then choose whether or not to run it. All Wizards available to the user would be automatically detected by PIY-II on start-up, so Wizards can be added to PIY-II in the same way much of the other areas of PIY-II can be extended.

8.2.3: User Interface enhancements

The GUI that a user can create using PIY-II is limited at the moment in that menus cannot be added to windows. This feature could feasibly be added to PIY-II by adding a new "JMenu" property to the UserWindow class, and by adding plug-in support for the JMenu type. However, adding some king of item listener to the menu might be needed - this would be hard to implement by a plug-in. So PIY-II may need to be slightly modified to accommodate this plug-in.

8.2.4: Rethinking PIY-II

Improvements to the flexibility of PIY-II may require a rethink on how it is currently structured. For example, there is currently a global memory store - that is, all variables that the user defines can be edited by any action at any time. It might be tidy to add local variable stores to each action list - that is, each action list (essentially a function) could have its own set of private variables to work on. This is a nontrivial enhancement to PIY-II, since the process of running an action list doesn't create a *copy* of the action list - so running an action list twice means that the "private" variables in the second run will be set to whatever the first run left them as. Perhaps a solution to this is clone the private variables such that whenever an action list is executed it first resets its variable's values to those of the clones. This is not quite as simple as it seems, since an action stores the reference to the original variable, not the clone. An action that chops a String in half, say, chops the original String rather than the clone.

If these problems could be overcome, PIY-II could begin to offer some true power. One particular feature that could be introduced would be a recursion mechanism. Here, action lists could take parameters and return values, and work on their own local as well global variables.

This is all perhaps beyond the scope of what PIY-II was initially intended to do. However, the current user interface wouldn't be significantly different to what it is now - the added power would effectively be working behind the scenes. The users would be receiving the extra features without having to worry about how to use them. Once they became more advanced PIY-II users, they could begin to take advantage of these features. I believe that local variables and recursion should be an integral part of PIY-II; if I'd had had more time to write PIY-II, these are the features I would have added. It is certainly not an easy goal to achieve however.

8.2.5: Compiler Optimisations

Currently, the PIY-II compiler creates a jar file which includes some specific PIY classes, along with all of the PIY-II components, containers, actions, etc. Not all of these classes are used by the user's application though - they are added regardless. Currently the number of these classes means that it's not a great overhead, but as more components are added to the PIY-II environment it could become more of an issue. There are various techniques that could be used to determine which classes are needed for a application to run, noting that some components rely on others for their operation. The simplest way I envisage it could be done would be to add a getDependencies method, which returns the array of classes that the component is directly reliant upon. Starting with the actual components the user has used in their project, a tree of all of the dependent classes could be built up. All of the classes in this tree could then be added to the jar file; none other are required.

Chapter 9: Conclusions

Although designed with teams in mind, XP can work equally well for a single programmer. Obviously a number of the practices need to be modified or dropped for them to make sense, but these will be the practices that are in place to help the management of teams of programmers.

There's no doubt that a lot of research is still needed to determine the true effectiveness of XP. The crux of the matter is that that *it does seem to work.* Some people still view XP as nothing more than a collection of common sense principles and no great feat, but the question has to be asked: If it's so simple, why has nobody thought of it before. Of course, they have. But they haven't built a methodology around it. It's somewhat of a mystery as to why it has taken so long to actually build software based on common sense. Management love paper, but programmers love code. The customer doesn't want paper.

I must admit that I'm still not completely sold on XP. It worked for me by not getting in the way, but there were times when I didn't know where to go. The process behind the practices is not very supportive - XP doesn't make any decisions for you, and offers very few hints when you get stuck. With regard to the 12 core practices, I can only conclude that not enough is known. Research has been done on pair programming, but more is needed.

On the whole, XP offers a good framework, but it needs to be tailored to fit different needs. It's difficult to know exactly what parts can be safely tailored in such a way - few of the practices are self reliant, and removing one means other corresponding practices are considerably weakened.

Of course, PIY-II has, in my opinion, been successful. PIY-II offers an extensible framework which is flexible enough to create many simple applications. Whilst more components and actions can be added to PIY-II in the future with relative ease, it may still benefit from a few more changes in structure to accommodate local variables in each action list. It could also do with a set of actions to control access to an underlying database, so that users could create basic database applications more easily. This particular enhancement to PIY-II can be done with a plug-in though, showing that PIY-II has been successful in its aim to be as extensible as possible.

The goal for PIY-II was to be able to compile and run projects - this goal has been met. The goal of analysing XP was to see how well it accommodate the process of constructing correct software. The artificial environment I have used the XP practices in doesn't really give any clear conclusions, apart from "it's too early to say". XP has used on projects which have succeeded, but it has been used on projects which have failed too. In XP's defence, the projects that have failed have tended to fail for either political reasons, or because certain XP practices were not adopted by the team.

Every project is different - some may benefit from using the XP approach, some may not. The real problem comes in choosing a methodology that is best suited to the problem at hand. No methodology should be taken on board statically - the team should be confident enough to be able to mould it to suit their problem domain. If the team cannot mould the methodology in this way, it is either because they are scared of it (not a good start), or because it is too rigid (most heavyweight methodologies fall into this category).

It may be that some projects are too complicated for the team not to follow a rigid methodology, but for small to medium sized projects, with volatile requirements and with a quick release version needed, XP seems to be a powerful and appropriate method.

It's worth remembering that I consider PIY-II to be a success. XP's aim of ensuring the customer is happy with the product developed was certainly realized, so long as it is me that is considered the customer.

Annotated Bibliography

[BECK99]. Beck, K., *Extreme Programming Explained - Embrace Change*, (Addison Wesley, 1999.)

The original book on XP, known in XP circles as the *white book*. Introduces the general philosophy behind XP - it is not particularly aimed at a technical audience, or at least it doesn't provide any real treatment of how to implement XP.

[BECK00]. Beck, K., Fowler, M., *Planning Extreme Programming*, (Addison Wesley, 2000.)

The third book released on XP, also known as the *green book*. It looks at how you should plan projects within an XP framework, looking specifically at the user stories, the planning game, short iterations, frequent releases and how to estimate risks and time.

[BROOKS95]. Brooks, F, P., *Mythical Man Month (20th anniversary edition)* (Addison Wesley, 1995.)

As well as containing the now infamous law known as Brooks' Law, this edition also contains the "No Silver Bullet" essay which was originally published in 1986, as well as an evaluation by Brooks as to what he got wrong (and right) in the original edition.

[BROWN98]. Brown, W., et al., *Anti-Patterns*, (John Wiley and Sons, 1998)

An anti-pattern is a common error or mistake that people make in designing object-oriented systems. This book describes how to detect anti-patterns in code, and how to refactor code to remove these anti-patterns.

[COCKERILL98]. Cockerill, J., *Program It Yourself*, (3rd year dissertation project, Department of Computer Science, University of Sheffield, 1998.)

One of the original PIY projects undertaken. It introduces the interesting idea of attaching pre-programmed functions to certain application events - an idea that forms the basis of PIY-II.

[COOPER00]. Cooper, J, W., *Java Design Patterns*, (Longman Higher Education, 2000)

Describes the design patterns from the original *Design Patterns* book, but with implementations in Java. The style here is more laid back than *Design Patterns* - concentrating on describing the patterns rather than the theory behind them.

[FOWLER99]. Fowler, M., *Refactoring: Improving the Design of Existing Code*, (Addison Wesley, 1999.)

The refactoring bible. This book aims to be to refactoring what the GOF book is to design patterns. It is certainly a comprehensive collection of simple refactorings, but it could be argued that the majority of refactorings are too simple to justify their inclusion.

[FOWLER00]. Fowler, M., *The New Methodology*, (Online article, available at www.matinfowler.com/articles/new/newMethodology.html, 2000)

This article gives an overview and assessment of the new styles of methodologies that have recently gained popularity (the lightweight methodologies).

[GAMMA94]. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, (Addison Wesley, 1994)

The original book on design patterns, containing a catalogue of patterns aimed at solving particular object-oriented system design problems

[HOLCOMBE98]. Holcombe, M., *Correct Systems - Building Business Process Solutions*, (Springer 1998.)

Defines what it means for a system to be "correct", as explores how to go about testing systems using X-Machines.

[HOLCOMBE01-1]. Holcombe, Gheorghe, Macias, *Teaching XP for real: Initial Observations and Plans* (University of Sheffield, 2001.)

Introduces an experiment carried out on XP to see what advantages it could have over traditional software development methodologies.

[HOLCOMBE01-2]. Holcombe, Bogdanov, Gheorghe, *Functional Test Generation for Extreme Programming* (University of Sheffield, 2001.)

This paper first asserts that systems must be designed for test. From there, test sets which offer total cover of the system can be developed in a formal way using X-Machines.

[INCE95]. Ince, D. C., *An Introduction to Discrete Mathematics, Formal System Specification and Z* (second edition), (Oxford University Press, 1995.)

An introduction on how to specify systems formally using Z, as well as some introductory chapters defining what a "correct" system is.

[JEFFRIES00]. Jeffries, R., et al., *Extreme Programming Installed* (Addison Wesley, 2000)

The second book on XP, also known as the *pink book*. It acts as a guide to implementing the core practices in XP, and goes into more technical detail than Beck's *Extreme Programming Explained*. That said, coverage of many of the practices is not really technical enough - there's a lot of justification rather than demonstration.

[JEFFRIES01]. Jeffries, R., *xProgramming.com* (Website, at www.xProgramming.com, 2001.)

This website contains quite a few articles and discussions on XP topics. Very biased. Doesn't contain any articles attacking XP.

[MCMINN99]. McMinn, P., *Program It Yourself*, (3rd year dissertation project, Department of Computer Science, University of Sheffield, 1999.)

The PIY application on which PIY-II has been partially based. Much of the groundwork used in PIY-II with regards to requirements was taken from this dissertation - I have tried to concentrate on implementation rather than requirements capture. This dissertation also includes an analysis of testing using X-Machines, which contrasts with the method of unit tests used for PIY-II.

[MYERS79]. Myers, G., *The Art of Software Testing*, (Wiley-Interscience, 1979.)

Covers various methods of testing programs (non object-oriented). Shows how tests cannot prove the absence of bugs, contrary to the XP premise that unit tests and functional tests suffice.

[OPDYKE92]. Opdyke, W. F., *Refactoring Object-Oriented Frameworks*, (Ph.D. thesis, University of Illinois, 1992.)

The first thorough treatment of refactorings as behaviour preserving transformations. Opdyke proves that the refactorings he propose preserve some predetermined invariants. Opdyke does not prove that preserving these invariants preserves program behaviour though.

[OVERTON98].Overton, M., *Program It Yourself*, (3rd year dissertation project, Department of Computer Science, University of Sheffield, 1998.)

One of the original PIY projects.

[WILLIAMS00]. Williams, Kessler, Cunningham, Jeffries, *Strengthening the Case for Pair Programming*, (IEEE Software, July/August 2000, pages 19-25.)

Article which reports on results obtained from research that has been carried out on pair programming, giving evidence to the claim that two programmers working together on the same task produce better quality code in a shorter amount of time.

[XPMAP00]. *Extreme Programming Roadmap* (Mirrored from http://c2.com/cgi/wiki on 23/Sep/00.)

This extensive website covers all aspects of XP, and since it is a "wiki" site, anyone viewing the site can edit its contents. This makes the site very up-to-date, but sometimes quite difficult to follow - the text leads on to many unrelated topics. That said, virtually every facet of XP is covered in detail.

Appendix A: Glossary of Terms

Extreme Programming introduces quite a few new terms that aren't generally recognised outside of the methodology - indeed, there is a culture of naming practices in certain ways to match their names on the XP wiki site (see XPMAP00), or to name the XP books in terms of their colour (ie. "the white book", or "the pink book", etc.) This brief glossary introduces the more commonly used terms.

Acceptance Test	The now common name for functional tests - the tests created by the customer to check that particular stories have been implemented correctly.
Agile Method	This term is gaining more usage - an alternative to "Lightweight Methodology".
BDUF	Big Design Up Front - a part of the Waterfall method
BUFD	Big Up Front Design - ie. BDUF
DRY	"Don't Repeat Yourself" - from the *Pragmatic Programmer*, similar in ways to "Once and only once".
Do The Simplest Thing That Could Possibly Work	When implementing a new feature, don't try and create a large design or abstract class hierarchy for it - implement in the simplest way that does everything that is needed to be done.
DTSTTCPW	Acronym for "Do the simplest thing that could possibly work".
Gang Of Four	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides - authors of the *Design Patterns* book. (See GAMMA94)
GOF	Acronym for Gang of Four
Green Bar	A term which means that all of the unit tests have passed. The term originates from JUnit, which displays a green bar on passing all the tests.
Green Book	*Planning Extreme Programming* (see BECK00).
OAOO	Acronym for "Once and only once".
Once and only once	Term meaning that the code says precisely what it needs to say just once - a premise of simple design.
Pair Programming	Two programmers working side-by-side on the same design, algorithm, code or test.
PEP	Acronym for *Planning Extreme Programming* (See BECK00).
Pink Book	*Extreme Programming Installed* (See JEFFRIES00).
PP	Acronym for Pair Programming.
RUP	Acronym for the Rational Unified Process.
Smalltalk	The programming language used on the C3 project - a particular favourite of Kent Beck.
Spike	A quick solution to a problem, intended for research. That is,

	the spike solution tests how a particular solution might work, before a specific solution is committed to.
Test Everything That Could Possibly Break	The idea that unit tests should test everything that the programmer fears may break - generally said in response to the question “what should I write unit tests for?”
Unit Test	An automated test that tests a single class or collection of classes. Usually implemented in one of the xUnit series.
UT	Acronym of Unit Test.
White Book	*Extreme Programming Explained* (See BECK99).
XPE	Acronym for *Extreme Programming Explained* (See BECK99).
XP Values	Communication, Simplicity, Feedback and Courage.
xUnit	A series of testing frameworks that provide a way for programmers to write and manage automated unit tests in practically any programming environment.
YAGNI	Acronym for You Aren’t Gonna Need It.

Appendix B: Discussions of XP Concepts

This section is intended to serve as an aside to the main topics discussed in the project. Its main aim is to highlight a few of the misconceptions that people have about XP, and to give further justification to some of the core XP practices.

XP Design and Refactoring

If XP has no analysis phase, when is design considered?

Design is one of the areas in which XP is attacked the most. Does XP advocate a code and fix strategy? Of course not - design is intended to be evolutionary in XP. Design in XP consists of localised design decisions based on solving particular design problems at the time

There *is* an initial design though - a structure to work towards, given in the system metaphor. However, this is not nearly detailed enough to explain the entire design of the system. The programmers need to make small scale design decisions whenever a new feature is to be added. There are two places in which a new feature can be added:

1. Add a new class to the system to support the new feature;
2. Put the new feature in an existing class.

But what happens when neither of the above are desirable, or even possible? That is, what happens when the new feature impacts existing classes in a way which has not yet been designed for? Traditional design approaches attempt to prevent this from happening. The idea that design is something done at the start of a project, and is not subject to change is something the XP disregards completely. XP believes that designing at the point of implementation is a better technique than designing at the point of requirements conception. To keep the design created in this way clean, XP uses refactoring - the act of restructuring code whilst retaining its function.

Design Patterns and XP

Design Patterns + YAGNI = Nothing?

The quickest way to confuse an XP advocate is to ask them to comment on the above equation. Can design patterns coexist with YAGNI?

The quickest (and possibly safest) answer is *maybe*. The very idea of design patterns is that they are just that - *designs*. They are intended to solve a particular design problem. However, design patterns encompass the whole design solution - YAGNI asserts that you should not add code that isn't needed. The point is really as to whether the overuse of patterns in contrary to YAGNI's advice. In using a particular pattern, do you really need all of the functionality it offers. Compare this with the fact that patterns have been proven to work - should you change a pattern to make it simpler just to be in keeping with YAGNI?

When discussing the use of design patterns in an XP project, the example usually considered is JUnit - the unit testing framework written for Java by Kent Beck and Erich Gamma (one of

the four authors of the *Design Patterns* book [GAMMA94]). The source code for JUnit shows the high density of patterns used in the design, which is hardly surprising for a program written by Erich Gamma, but since Kent Beck wrote it too, it does has some significance as to the place design patterns serve within an XP environment. When it is realised that Kent Beck is (was) an active patterns advocate, and has gone as far as writing a patterns book, Smalltalk best practice patterns, it becomes even stranger that patterns seem to be discouraged in XP. Ward Cunningham, considered a co-inventor of XP with Beck, has also been very active in the patterns community; XP grew out of the patterns community. So where are patterns in XP? The short answer is *Refactoring*. When code is refactored a few times, the resulting article often contains the design patterns as a consequence. That is, refactoring can be thought of as a small step towards a pattern. The pattern is the complete solution, the refactoring is a single step towards realising that solution. One advantage of refactoring is that it's more obvious (some patterns seem strange, and often don't really explain the how or why). Another advantage is that you aren't forced to implement a lot of functionality in one go, as a pattern generally requires. Refactorings don't change the function of the code - but they can make the code more amenable to change. Patterns aren't the enemy - they are useful, but it's sometimes difficult to know when.

Whether you should attempt to refactor code with the aim of approaching a particular pattern is another question entirely. Refactorings give rise to an emergent design - the more "natural" design to *accommodate* the code. Trying to guide this process seems to be against the grain. In the words of Kent Beck: "Start stupid and evolve".

Appendix C: PIY-II User Stories

The user stories are a collection of requirements for the system to be developed. Each one should be free of technical information and only go into enough detail that a low risk estimate can be made on the time it will take to implement the story.

A story will be typically three lines in length and should be written on individual cards. Between 60 and 80 stories is a good number to start the planning game with.

To avoid confusion, users of PIY-II will be referred to as "users", and user of applications developed by PIY-II will be known as "end-users".

Note that the stories given here are the *initial* user stories that were drawn up for PIY-II. The requirements of the system have changed since these stories were written - what seemed important at the time turned out not to be so, and the requirements have changed over time to reflect this.

Some of these stories are also too technical, and do not serve as a good example set of user stories. Customers cannot be expected to be perfect, however.

Example stories from PIY-II

Loading/saving and importing projects

1. Projects can be loaded and saved, as well as renamed from within PIY-II.
2. Only one project can be loaded into PIY-II at any one time.
3. If the user attempts to load a project when an unsaved project is currently open, the user should be presented with the options a) save the current project; b) load the new project anyway; or c) Cancel - continue editing the current project.
4. If the user attempts to save the current project with a filename that is currently in use, the user has the options a) Overwrite the file; b) Choose a different filename; or c) Cancel the save operation - ie. don't save.
5. A PIY-II project should be able to be imported into another PIY-II project. Any clashes in names should be automatically resolved by PIY-II, if necessary by changing the names of the components/action lists that have been imported.

GUI editor / Components

6. There must be a GUI editor in which the user can add/remove or modify components on the user windows.
7. The GUI editor window must contain a section to edit properties of the GUI component, and a section to edit the actions attached to the events associated with the component.
8. When a user selects a component on their GUI, the GUI editor window should show the properties of that component.
9. Every component on the GUI has a unique name which the user can change to a different unique name.
10. No component can have the name NULL.

11. If the user deletes a component from their GUI, all actions that reference the component must now reference NULL instead.
12. A component on one window must have a different name to components on all other windows. Any action can reference a component on any window via its unique name.
13. If the user changes the name of a component, any of the actions that refer to that component must be automatically updated to reflect the change.
14. Every component and action provided by PIY-II must have a description associated with it. The description must be shown (or easily accessible) to the user when the component/action has been selected.

Action List editor / Actions

15. An action list editor must be present where users can add/remove/edit actions on an action list.
16. When a user selects an action in the action list editor window, another window should display the properties of that action.
17. Users can attach actions to any events that can occur in their applications. Each action the user can attach must have some form of visual representation and description to the user - no code can be shown.
18. Users can attach a sequence of actions (an action list) to a specific component event.
19. Users can create an action list that is not attached to any events. Each action list not attached to an event must have a (unique) name, which can be modified by the user.
20. Actions can be added to PIY-II by the user.
21. If the user deletes a named action list, all other action lists that call that action list must now reference NULL.
22. The user can attach, to any action list, a textual description of what the action list is attempting to achieve.
23. If the user changes the name of an action list, any other actions that call the action list must be automatically updated to reflect the change.

Running/Testing

24. Users must be able to run and test their projects from within PIY-II.
25. If the user's program is being run from within PIY-II, and it comes across a reference to NULL, it must terminate and show the user where the NULL reference is.
26. If a project is being run from within the PIY-II environment, it should be possible for the user to terminate the project from PIY-II.

General

27. Users can modify the position/size of a component on their GUI by using the GUI editor window or by modifying the component's properties in the component property window.
28. Users can create as many windows as they like, and have them all open at the same time for editing (if they wish).
29. There must be an online help system accessible to the user at all times.
30. Every component and action provided by PIY-II must have an associated example that shows the component/action in use.

Constraints

31. Components must include buttons, radio buttons, combo-boxes and lists.
32. There must be database component in which records can be displayed and edited in the GUI by an end-user.
33. Events must include “Clicked” - called when a particular button has been clicked.
34. There must be an “if...then...else” action. It will consist of one Boolean expression and two action lists.
35. A “calculation” action must be present in PIY-II. The calculation action should allow users to perform a calculation based (perhaps) on data stored in the current GUI, and send the result of the computation to a component on the GUI.
36. There must be an action to call another action list.

Appendix D: PIY-II Functional Tests

A functional test should be present for every functional requirement in every story. Kent Beck [BECK99] writes:

> *The question they [the customers] need to ask themselves is "What would have to be checked before I would be confident that this story is done?" Each scenario they come up with turns into a test, in this case a functional test.*

Example functional tests for PIY-II

The functional tests given here are as the customer would write them - they still need to be converted to code so that they can be automated and run against PIY-II. The tests correspond to the *initial* user stories - whenever a user story is changed, the relevant test needs to be changed as well.

Loading/saving and importing projects

Ref	Test	Expected Outcome
1.1	Load valid PIY-II project	Project loads
1.2	Load invalid (non-PIY II) project	Error message – project doesn't load.
1.3	Save current PIY-II project that has a filename	Project saves
1.3.1	Reload saved project	The open project is the same as the one that was saved.
1.4	Rename a project - save it under a different filename	User should be asked for a filename
1.4.1	Perform tests 4.1, 4.2	Project should be saved under a different filename
1.5	Save a current PIY-II project that doesn't have a filename	User should be asked for a filename
1.5.1	Perform tests 4.1, 4.2	Project should be saved under a different filename
2.1	Load a project with a project currently open.	Either a) the new project is opened; or b) the current project remains open.
3.1	Load valid project with an unsaved project open	Dialog should appear with the option to save the current project, or load the new project anyway, or to cancel the load operation.
3.1.1	Perform tests 4.1, 4.2	Project should be loaded
3.1.2	Don't save the unsaved project	Project should be loaded
3.1.3	Cancel the load operation	Project should not be loaded, current project remains.
3.2	Load a project with a saved project currently open	Project should be loaded.

4.1	Save the current project under a different filename that already exists	Dialog should appear with the options a) Overwrite the file; b) Choose a different filename; c) Cancel the save operation.
4.1.1	Choose overwrite	Project saved under different filename.
4.1.1.1	Reload project that was just saved.	Project loads and is identical to the one just saved.
4.1.2.1	Choose a different filename that exists	As for test 4.1
4.1.2.2	Choose a different filename that doesn't exist	Project should be saved
4.1.3	Cancel the save operation	Project should not be saved - the file with the filename that was in use should remain untouched.
4.2	Save the project under a different filename that doesn't exist.	Project should be saved.
5.1	Import a project without any name clashes with the current project.	The names of components/action lists should be unchanged in both the imported code and the current project.
5.2	Import a project with only one component name clash with the current project.	The component names should be unchanged, apart from the name that clashes - which should only be changed for the imported project.
5.3	Import a project with only one action list name clash with the current project.	The action list names should be unchanged, apart from the action list with the name that clashes, which should only be changed for the imported project.
5.4	Import a project in which all of the component names clash with the current project.	All the component names in the imported project should have been changed, none in the current project should be changed.
5.5	Import a project in which all of the action-list names clash with the current project.	All the action list names in the imported project should have been changed, the current project action list names should be unchanged.

GUI Editor / Components

Ref	**Test**	**Expected Outcome**
9.1	Change the name of a component to a unique name	Name of component is changed
9.2	Change the name of a component to an non-unique name	Dialog appears asking for a different name
9.2.1	Enter a unique name	Name of component is changed
9.2.2	Enter a non-unique name	Dialog appears asking for a different name
9.2.3	Press cancel	Component's previous (unique) name is kept
10.1	Change the name of a component to NULL	Dialog appears asking for a different name

10.1.1	Enter NULL again	Dialog appears asking for a different name
10.1.2	Perform tests 9.1, 9.2	Component name becomes a new unique name, or previous (unique) name is kept
11.1	Delete a component that has no actions referencing it	All actions should remain the same as they did before the delete
11.2	Delete a component that has one action referencing it	The referencing action should reference NULL for the deleted component; all other actions should remain unchanged.
11.3	Delete a component that has more than one but less than all actions referencing it	All referencing actions should reference NULL for the deleted component; all other actions should remain unchanged.
11.4	Delete a component that has all actions referencing it	All actions should reference NULL instead of the deleted component – all other references should remain unchanged.
12.1	Change the name of a component to a name unique on all windows.	Name is changed
12.2	Change the name of a component to a name that is not unique on all windows	Dialog appears asking the user to change the name
12.2.1	Perform test 9.2	Name becomes a name unique on all windows, or component's previous unique name is kept.
13.1	Change the name of a component that is not referenced in any action	All actions remain unchanged.
13.2	Change the name of a component that has one action referencing it	The referencing action now references the changed component name – all other references and other actions remain unchanged
13.3	Change the name of a component that has more than one but less than all actions referencing it	All of the referencing actions reference the changed component name – all other references and actions remain unchanged
13.4	Change the name of a component that has all actions referencing it	All actions reference the changed name, but all other references remain unchanged.

Action List editor / Actions

Ref	**Test**	**Expected Outcome**
19.1	Change the name of an action list to a unique name	Action list name is changed to the new name
19.2	Change the name of an action list to an non-unique name	Dialog appears asking for a different name
19.2.1	Enter a unique name	Action list name is changed to the new name
19.2.2	Enter a non-unique name	Dialog appears asking for a different name
19.2.3	Cancel the change name operation	Name of action list is restored to its previous unique name

20.1	Change the name of an action list to NULL	Dialog appears asking for a different name
20.2	Enter NULL again	Dialog appears asking for a different name
20.3	Perform test 19.2	Action list name becomes a different unique name, or keeps its previous name, but isn't NULL.
22.1	Delete a named action list that is not referenced by any other action lists	All action list references remain unchanged
22.2	Delete a named action list that is referenced by one other action list	The action list reference now references NULL, all other action lists remain unchanged
22.3	Delete a named action list that is referenced by more than one other action lists, but not by all	All action list references now reference NULL, all remaining action lists remain unchanged
22.4	Delete a named action list that is referenced by all other action lists	All action lists reference NULL, but all other references remain unchanged.
24.1	Change the name of an action list that is referenced by no other action lists	All other actions lists remain unchanged
24.2	Change the name of an action list that is reference by one other action list	The action list now references the changed name, all other action lists remain unchanged
24.3	Change the name of an action list that is referenced by more than one but less than all other action lists	All referencing action lists reference the new name, but all remaining action lists remain unchanged
24.4	Change the name of an action list that is referenced by all other action lists	All action lists reference the new name, all other references remain unchanged
24.5	Change the name of an action list that references itself	The action list should reference itself under its new name

Running / Testing

Ref	**Test**	**Expected Outcome**
26.1	Run a program with no NULL references	Program doesn't terminate due to NULL reference
26.2	Run a program with one NULL reference that is never reached	Program doesn't terminate due to a NULL reference
26.2	Run a program with a NULL reference that is reached	Program terminates due to a NULL reference

Appendix E: Class Diagram Notation

The notation used throughout for UML diagrams is not quite that which is in standard usage - the notation used is a slightly modified form of that used in [GAMMA94]: