All you ever wanted to know about Patterns,

but were afraid to ask!

John Hunt
JayDee Technology Limited
Minerva House
Bath BA2 9ER
Telephone: +44 (0)1225 789255
Fax: +44 (0)870 0548872
Email: John.hunt@jaydeetechnology.co.uk

Abstract

This paper provides an introduction to design patterns. Design patterns describe both the design of some thing as well as that thing. In the case of software, a design pattern often describes the software and how to use / apply that software in different situations. This paper attempts to present the motivation behind design patterns (often referred to as just patterns), what they are (and what they are not) and when they are useful. An example (coded in Java) is then included to illustrate the concept of a Pattern.

1. Introduction

There is a growing interest in what has become known generically as patterns. To be more precise in Design Patterns. This is evidenced by the two articles in the Winter 1996 issue of the OOPS Newsletter. One of these articles in particular presented a very good introduction to the philosophy of patterns [Birch 1996]. However, neither article attempted to present a sample pattern, to highlight the many motivations behind the uptake of patterns or consider when they should be used. This paper attempts to provide these along with a brief consideration of the strengths and weaknesses of patterns.

Historically, design patterns have their basis in the work of an architect who designed a language for encoding knowledge of the design and construction of buildings [Alexander *et al* 1977, Alexander 1979]. The knowledge is described in terms of patterns that capture both a recurring architectural arrangement and a rule for how and when to apply this knowledge. That is, they incorporate knowledge about the design as well as the basic design relations.

This work was picked up by a number of researchers working within the object oriented field. This then led to the exploration of how software frameworks can be documented using (software) design patterns (for example, [Johnson 1992] and [Birrer and Eggenschmiler 1993]). In particular Johnson's paper describes the form that these design patterns take and the problems encountered in applying them.

Since 1995 and the publication of the "Patterns" book by the *Gang of Four* [Gamma *et al*, 1995], interest in patterns has mushroomed. Patterns are now seen as a way of capturing expert and design knowledge associated with a system architecture to support design as well as software reuse. In addition as interest in patterns has grow their use, and representational expressiveness has grown.

The remainder of the paper is structured in the following manner: Section two considers the motivation behind the patterns movement, section three considers what a pattern is and is not. Previous articles in the OOPS Newsletter have already introduced the concept of patterns [Birch 1996] and [Henney 1996]. Thus in this paper, only a brief introduction to the concept will be given and greater emphasis will be placed on describing, documenting and illustrating patterns. Section four describes how patterns are documented. Section five briefly considers when patterns should be used and section six discusses the strengths and limitations of patterns. Section seven then presents an example pattern using Java. This pattern considers how you can use a mediator along with a set of associated objects to communicate in a loosely coupled manner.

2. Motivation behind Patterns

There are a number of motivations behind design patterns. These include:

1. Designing reusable software is difficult. Finding appropriate objects and abstractions is not trivial. Having identified such objects, building flexible, modular, reliable code for general reuse is not easy, particularly when dealing with more than one class. In general, such reusable "frameworks" emerge over time rather than being designed from scratch.
2. Software components support reuse of code but not the reuse of knowledge.
3. Frameworks support reuse of design and code but not knowledge of how to use that framework. That is, design trade-offs and expert knowledge are lost.
4. Experienced programmers do not start from first principles every time; thus, successful reusable conceptual designs must exist.
5. Communication of such "architectural" knowledge can be difficult as it is in the designers head and is poorly expressed as a program instance.
6. A particular program instance fails to convey constraints, trade-offs and other non-functional forces applied to the "architecture"

7. Since frameworks are reusable designs, not just code, they are more abstract than most software, which makes documenting them more difficult. Documentation for a framework has three purposes and patterns can help to fulfill each of them. Documentation must provide:

   - the purpose of the framework,
   - how to use the framework,
   - the detailed design of the framework.

8. The problem with cookbooks is that they describe a single way in which the framework will be used. A good framework will be used in ways that its designers never conceived. Thus, a cookbook is insufficient on its own to describe every use of the framework. Of course, a developer's first use of a framework usually fits the stereotypes in the cookbook. However, once they go beyond the examples in the cookbook, they need to understand the details of the framework. However, cookbooks tend not to describe the framework itself. But in order to understand a framework, you need to have knowledge of both its design and its use.
9. In order to achieve high level reuse (i.e. above the level of reusing the class set) it is necessary to design with reuse in mind. This requires knowledge of the reusable components available.

The design patterns movement wished to address some (or all) of the above in order to facilitate successful architectural reuse. The intention was thus to address many of the problems which reduce the reusability of software components and frameworks.

3. Design Patterns

3.1 What are design patterns?

A design pattern captures expertise describing an architectural design to a recurring design problem in a particular context [Gamma *et al* 1993; Johnson 1992; Beck and Johnson 1994]. It also contains information on the applicability of a pattern, the trade offs which must be made, and any consequences of the solution. Books are now appearing which present such design patterns for a range of applications. For example, [Gamma *et al*, 1995] is a widely cited book which presents a catalog of 23 design patterns.

Design patterns are extremely useful for both novice and experienced object oriented designers. This is because they encapsulate extensive design knowledge and proven design solutions with guidance on how to use them. Reusing common patterns opens up an additional level of design reuse, where the implementations vary, but the micro-architectures represented by the patterns still apply.

Thus, patterns allow designers and programmers to share knowledge about the design of a software architecture. They thus capture the static and dynamic structures and collaborations of previous successful solutions to problems that arise when building applications in a particular domain (but not a particular language).

3.2 What they are not?

Patterns are not concrete designs for particular systems. This is because a pattern must be instantiated in a particular domain to be used. This involves evaluating various trade-offs or constraints as well as detailed consideration of the consequences. It also does not mean that creativity or human judgment has been removed as it is still necessary to make the design and implementation decisions required. Having done that the developer must then implement the pattern and combine the implementation with other code (which may or may not have been derived from a pattern).

Patterns are also not frameworks (although they do seem to be exceptionally well suited for documenting frameworks). This is because frameworks present an instance of a design for solving a family of problems in a specific domain (and often for a particular language). In terms of languages such as Smalltalk and Java a framework is a set of abstract, co-operating classes. To apply such a framework to a particular problem it is often necessary to customize it by providing user defined subclasses and to compose objects in the appropriate manner (e.g. the Smalltalk MVC framework). That is, a framework is a semi-complete application. As a result any given framework may contain one or more instances of multiple patterns and in turn a pattern can be used in many different frameworks.

4. Documenting Patterns

The actual form used to document individual patterns varies, but in general the documentation covers:

1. The motivation or context that the pattern applies to.
2. Pre-requisites that should be satisfied before deciding to use a pattern.
3. A description of the program structure that the pattern will define.
4. A list of the participants needed to complete a pattern.
5. Consequences of using the pattern, both positive and negative.
6. Examples of the patterns usage.

The pattern template used in [Gamma *et al*, 1995] provides a standard structure for the information which comprises a design pattern. This makes it easier to comprehend a design pattern as well as providing a concrete structure for those defining new patterns. Gamma’s book [Gamma *et al*, 1995] provides a detailed description of the template; only a summary of it is presented in Table 1.

Table 1: The design pattern template

Heading	Usage
Name	The name of the pattern
Intent	This is a short statement indicating the purpose of the pattern. It includes information on its rationale, intent, problem it addresses etc.
Also known as	Any other names by which the pattern is known.
Motivation	Illustrates how the pattern can be used to solve a particular problem.

Applicability	This describes the situation in which the pattern is applicable. It may also say when the pattern is not applicable.
Structure	This is a (graphical) description of the classes in the pattern.
Participants	The classes and objects involved in the design and their responsibilities.
Collaborations	This describes how the classes and objects work together.
Consequences	How does the pattern achieve its objective? What are the trade offs and results of using the pattern? What aspect of the system structure does it let you vary independently?
Implementation	What issues are there in implementing the design pattern?
Sample Code	Code illustrating how a pattern might be implemented.
Known uses	How the pattern has been used in the past. Each pattern has at least two such examples.
Related patterns	Closely related design patterns are listed here.

A pattern language is a structured collection of patterns that build on each other to transform needs and constraints into architecture. For example, the patterns associated with the HotDraw framework provide a pattern language for HotDraw. What is HotDraw? HotDraw is a drawing framework developed by Ralph Johnson at the University of Illinois at Urbana-Champaign [Johnson 1992]. It is a reusable design for a drawing tool expressed as a set of classes. However, it is more than just a set of classes; it possesses the whole structure of a drawing tool, which only needs to be parameterized to create a new drawing tool. It can therefore be viewed as a basic drawing tool and a set of examples that can be used to help you develop your own drawing editor!

Essentially HotDraw is a skeleton DrawingEditor waiting for you to fill out the specific details. That is, all the elements of a drawing editor are provided including a basic working editor, which you, as a developer, customize as required. What this means to you is that you get a working system much, much sooner and with a great deal less effort.

HotDraw was first presented at the OOPSLA'92 conference in a paper entitled "Documenting Frameworks using Patterns" by Ralph Johnson [Johnson 1992]. This paper is considers the problems associated with documenting complex reusable software systems using HotDraw as a concrete example. Included with the paper are a set of appendices which act as very useful guides on how to change the default drawing editor. The appendices represent HotDraw's pattern language and comprise ten different patterns. These ten patterns explain how to define drawing elements, change drawing elements, add constraints between graphic objects add lines etc.

I personally first used HotDraw in mid 1993 knowing nothing about patterns and didn't really understand the paper. However, I found the appendices helped me to customize the drawing editor quickly and painlessly. I read only those patterns I needed to understand what I wanted to do and ignored other patterns. Over time I found that I read those other patterns as and when I needed them.

5. When to use Patterns

Patterns can be useful in situations where solutions to problems recur but in slightly different ways. Thus, the solution needs to be instantiated as appropriate for different problems. The solutions should not be so simple that a simple linear series of instructions will suffice. In such situations patterns are overkill. They are particularly relevant when several steps are involved in the pattern which may not be required for all problems. Finally, patterns are really intended for solutions where the developer is more interested in the existence of the solution rather than how it was derived (as patterns still leave out too much detail).

6. Strengths and limitations of Design Patterns

Design patterns have a number of strengths including:

- providing a common vocabulary,
- explicitly capturing expert knowledge and trade-offs,
- helping to improve developer communication,
- promoting the ease of maintenance,
- providing a structure for change.

However, they are not without their limitations. These include:

- not leading to direct code reuse,
- being deceptively simple,
- easy to get pattern overload (i.e. finding the right pattern),
- they are validated by experience rather than testing,
- no methodological support.

In general, patterns provide opportunities for describing both the design and the use of the framework as well as including examples, all within a coherent whole. In some ways patterns act like a hyper-graph with links between parts of patterns. To illustrate the ideas behind frameworks and patterns the next section will present the framework HotDraw and a tutorial HotDraw pattern example explaining how to construct a simple drawing tool.

However, there are potentially very many design patterns available to a designer. A number of these patterns may superficially appear to suit their requirements, even if the design patterns are available on-line (via some hyper text style browser [Budinsky *et al* 1996]) it is still necessary for the designer to search through them manually, attempting to identify the design which best matches their requirements.

In addition, once they have found the design that they feel best matches their needs, they must then consider how to apply it to their application. This is because a design pattern describes a solution to a particular design problem. This solution may include multiple trade offs which are contradictory and which the designer must choose between, although some aspects of the system structure can be varied independently (although some attempts have been made to automate this process for example [Budinsky *et al* 1996]).

7. An Example Pattern: Mediator

This pattern is based on that present in [Gamma et al 1995] on pages 273-282. The Java code was written specifically for this paper and is available on the web at HTTP: //www.aber.ac.uk/~jjh/Java/Mediator.

Pattern name: Mediator

Intent: To define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly.

Motivation: Object oriented design encourages the distribution of behavior among objects. However, this can lead to a multiplicity of links between objects. In the worst case every object needs to know about / link to every other object. This can be a problem for maintenance and for the reusability of the individual classes.

These problems can be overcome by using a mediator object. In this scheme other objects are connected together via a central mediator object in a star like structure. The mediator is then responsible for controlling and coordinating the interactions of the group of objects.

Applicability: The mediator pattern should be used where:

- a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
- reusing an object is difficult because it refers to, and uses, many other objects.
- a particular behavior is distributed amongst a number of classes and we wish to customize that behavior with the minimum of subclassing.

Structure: The class diagram for a mediator is illustrated in Figure 1.

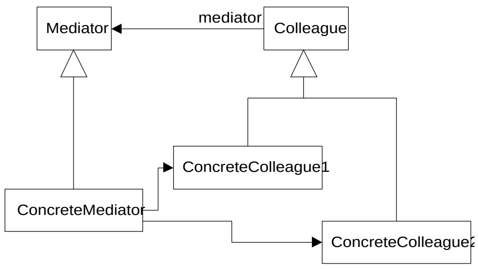


Figure 1: Mediator class diagram

A typical object diagram for a mediator is illustrated in Figure 2.

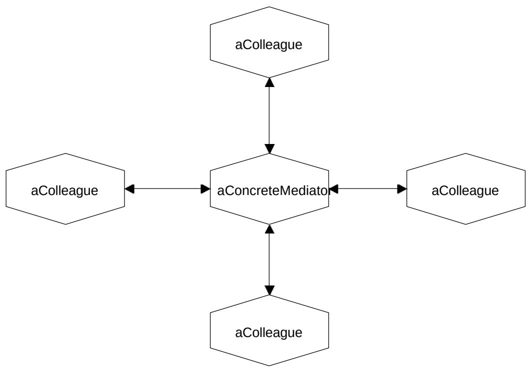


Figure 2: A mediator object diagram

Participants:

- **Mediator** handles communication between colleague objects.
- **ConcreteMediator** defines how the mediator should coordinate the colleagues interactions. It knows and maintains its colleagues.
- **Colleague classes** Each colleague knows its mediator object. It communicates with this mediator object in order to communicate with other colleagues.

Collaborations: The mediator object receives messages from colleagues and relays them to other colleagues as appropriate.

Consequences: The mediator pattern has the following benefits and drawbacks:

1. It limits subclassing to the mediator (e.g. by changing the routing algorithm in mediator you can change the systems behavior).
2. It de-couples colleagues.
3. It simplifies object protocols from many to many down to one to many.
4. It abstracts how objects cooperate.
5. It centralizes control.

Implementation: The following implementation issues are relevant to the mediator pattern:

1. Omitting the abstract Mediator class. If there is only to be one mediator class there is no reason to define an abstract class.
2. Colleague mediator communication. The colleagues need to tell the mediator when something interesting happens to them that they wish to relay to their colleagues. This could be handled via a dependency mechanism (see the Observer pattern) or by direct communication by the object. For example the colleague could tell the mediator that something has changed and then allow the mediator to interrogate it to find out what. This is the approach taken in the sample code example.

Sample code: The following illustrates the basic structure for the classes used in a simple mediator based system. The assumption used is that when colleagues need to communicate with the mediator, a colleague passes itself as an argument, allowing the mediator to identify the sender. The Mediator class (in Java) is:

```
import java.util.*;

public abstract class Mediator {
   private Vector colleagues = new Vector();
   public void addColleague(Colleague col) {
      colleagues.addElement(col);
   }
   public abstract void changed (Colleague col);
}
```

Concrete subclasses of Mediator (such as CommunicationManager) implement the changed method to affect the appropriate behavior. The colleague passes a reference to itself as an argument to the changed method to let the mediator identify the colleague that changed.
Colleague is the abstract class for all colleagues. A Colleague knows its mediator.

```
public abstract class Colleague {
   private Mediator mediator;
   public void addMediator(Mediator med) {
         mediator = med;
   }
   private void changed() {
         mediator.changed(this);
   }
}
```

As an example, consider an application in which we wish to inform members of a software team whenever a meeting has been arranged (we will ignore the issue of checking that all the team members can make that meeting). Rather than construct a rigid set of links between the members we will use the Mediator pattern. We can then define a CommunicationsManager class (which inherits from Mediator) as follows:

```
import java.util.Enumeration;

public class CommunicationsManager extends Mediator {
   public static void main (String args []) {
      CommunicationsManager c = new CommunicationsManager();
      c.setup();
      c.sampleMeeting();
   }

   public void setup () {
      int i;
```

```
    String teamMembers [] = {"John", "Denise",
                                                   "Phoebe", "Isobel"};
    for (i = 0; i < teamMembers.length; ++i) {
      addColleague(new TeamMember(teamMembers[i]));
    }
  }

  public void sampleMeeting () {
       // This is just an example the manager would
       // not normally initiate this.
       TeamMember aPerson;
       aPerson = (TeamMember)colleagues.firstElement());
       aPerson.meeting("9:00am 10/3/97");
  }

  public void changed (Colleague person) {
    TeamMember item;
    String theMeeting =
                          ((TeamMember)person).currentMeeting();
    for (Enumeration e = colleagues.elements();
                                                   e.hasMoreElements(); ) {
      item = (TeamMember)e.nextElement();
      if (item != person)
        item.newMeeting(theMeeting);
    }
  }
}
```

This class sets up the colleagues to be linked to the CommunicationsManager. In this case the colleagues are all instances of a class TeamMember (see below). It then uses an example method to trigger off communications between the teamMember objects. To achieve this the CommunicationsManager implements its own changed() method. This method merely passes details of the current meeting onto the other team members.

The TeamMember class extends the Colleague class and (for this simple example) can be defined as:

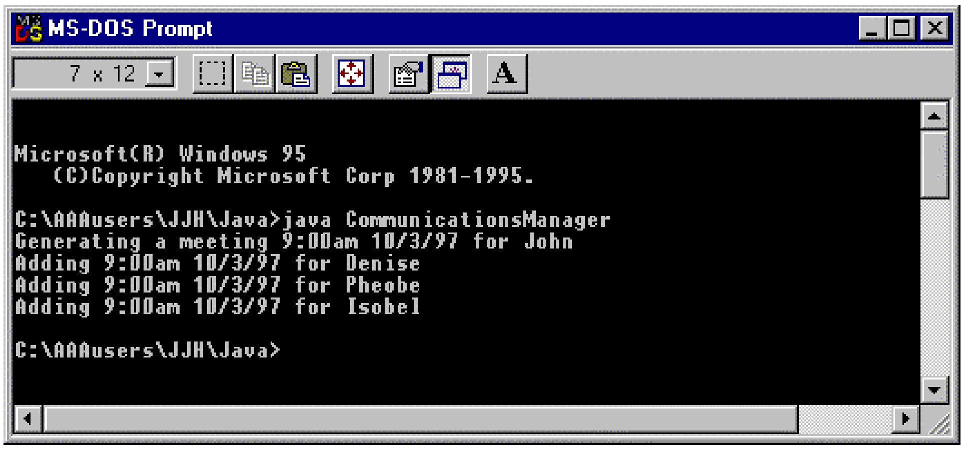
```
import java.util.Vector;

public class TeamMember extends Colleague {
  // Instance variables
  String name, meeting;
  Vector meetings = new Vector();
  // A Constructor
  public TeamMember (String aName) {
    name = aName;
  }
  public void meeting(String aTimeAndDate) {
    meeting = aTimeAndDate;
```

```
        System.out.println("Generating a meeting " +
                    aTimeAndDate + " for " + name);
        changed();
    }
    public String currentMeeting () {
        return meeting;
    }
    public void newMeeting (String aTimeAndDate) {
        meetings.addElement(aTimeAndDate);
        System.out.println("Adding " + aTimeAndDate +
                    " for " + name);
    }
}
```

This class defines the functionality of the TeamMember objects. It inherits all the functionality it needs to work with any form of mediator. The only detail that needs to be incorporated is a call to the changed() method when appropriate (in this case in method meeting()).

An example of this application running is presented in Figure 3.



```
Microsoft(R) Windows 95
   (C)Copyright Microsoft Corp 1981-1995.

C:\AAAusers\JJH\Java>java CommunicationsManager
Generating a meeting 9:00am 10/3/97 for John
Adding 9:00am 10/3/97 for Denise
Adding 9:00am 10/3/97 for Pheobe
Adding 9:00am 10/3/97 for Isobel

C:\AAAusers\JJH\Java>
```

Figure 3: The CommunicationsManager application

Known uses: ET++ and the THINK C class library use director like objects in dialogs as mediators between widgets. Smalltalk/V uses it as the basis of its application architecture.

Related patterns: Facade differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. However its protocol is unidirectional where as mediator is multi-directional. Colleagues can communicate with a mediator using the Observer pattern.

8. Summary

In this paper we explored the concepts of patterns as a method of documenting the design of reusable software architectures. Such patterns have a great deal of potential, however on-line support for browsing and applying patterns is required. In addition work on methodologies which consider how to define and apply patterns is required.

9. Further Reading

A number of books and a great many papers have been written about patterns in recent years. The most influential of which is [Gamma *et al* 1994] by the so called “Gang of four” who are Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. There are also a series of conferences on Patterns referred to as PLoP (for **P**attern **L**anguage **o**f **P**rogram design). Two proceedings are available [Coplien and Schmidt 1995] and [Vlissides *et al* 1996].

Two further patterns books are [Buschmann *et al* 1996] (which represents the progression and evolution of the pattern approach into a system capable of describing and documenting large scale applications) and [Fowler 1997] which considers how patterns can be used for analysis to help build reusable object models.

In addition to the papers mentioned earlier in this paper, there is also a web page dedicated to the patterns movement (which includes many of the papers referenced as well as tutorials and example patterns). The URL for the web page is: http://st-www.cs.uiuc.edu/users/patterns/patterns

10. References

[Alexander et al 1977] C. Alexander, S. Ishikawa and M. Silverstein with M. Jacobson, I. Fiksdahl-King and S. Angel, *A Pattern Language*, Oxford University Press, 1977.

[Alexander 1979] C. Alexander, *The Timeless Way of Building*, Oxford university Press, 1979.

[Beck and Johnson 1994] K. Beck and R. Johnson, Patterns Generate Architectures, Proc. *Eccop’94*, pp. 139-149, Springer-Verlag, 1994.

[Birrer and Eggenschmiler 1993] Andreas Birrer and Thomas Eggenschwiler, “Frameworks in the Financial Engineering Domain: An Experience Report:, *ECOOP’93*, pp. 21-35.

[Budinsky *et al* 1996] F. J. Budinsky, M. A. Finnie, J. M. Vlissides and P. S. Yu, Automatic code generation from design patterns, *IBM Systems Journal*, Vol. 35, No. 2, 1996.

[Buschmann et al 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*, Wiley and Sons Ltd., ISBN 0-471-95869-7, 1996.

[Coplien and Schmidt 1995] J. O. Coplien and D. C. Schmidt (eds*), Pattern Languages of Program Design*, Addison-Wesley, ISBN 0-201-60734-4, 1995.

[Gamma et al 1993] E. Gamma, R. Helm, R. Johnson and J. Vlissades, Design patterns: Abstraction and reuse of object-oriented design, in *ECOOP’93 (Lecture Notes in Computer Science 707)*, pp. 406-431, Springer-Verlag, 1993.

[Gamma *et al*, 1995] E. Gamma, R. Helm, R. Johnson and J. Vlissades, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Fowler 1997] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, ISBN 0-201-89542-0, 1997.

[Johnson 1992] Ralph. E. Johnson, Documenting Frameworks with Patterns*, Proc. OOPSLA'92, SIGPLAN Notices* 27(10), pp. 63-76, 1992.

[Krasner and Pope 1988] G. E. Krasner and S. T. Pope, A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80, *JOOP* 1(3), pp. 26-49, 1988.

[Vlissides *et al* 1996] J. M. Vlissides, J. O. Coplien and N. L. Kerth, *Pattern Languages of Program Design 2*, Addison-Wesley, ISBN 0-201-89527-7, 1996.

They are feared across the galaxy. Their distinctive armor is recognizable from the Corellian Sector to the Outer Rim . They have acted as hired guns for the biggest, baddest dudes in the Empire. But how much do we really know about Mandalorians ? They donâ€™t really do much PR.Â If you know one thing about Mandalorian culture, itâ€™s probably that they are double-hard warrior folk who arenâ€™t likely to back down in a fight. Based on the Spartans, the Mandalorians prefer to let their fists do the talking, and frequently settle disputes with hand-to-hand combat. However, unlike the Spartans, the strong are rewarded but the weak arenâ€™t discarded â€" losers are treated with respect if they fought bravely. Are Russians afraid of the cold? â€œHow can you feel cold, arenâ€™t you Russian!?â€ Thatâ€™s probably one of the most annoying questions that Russians living abroad hear all the time.Â Eating them is considered to be low-class, and some Russians despise those who eat them, calling them gopniks or rednecks. But if you start, you canâ€™t finish until the seeds are all gone, everything around is covered in black shells. READ MORE: Sunflowers for Russians: Guilty pleasure and antistress measure. Alexander Legky/Global Look Press.Â Everything you ever wanted to know about Moscowâ€™s Kremlin. Everything you should know about the USSR. This website uses cookies. Click here to find out more. To be honest, the reason weâ€™ve refrained from posting much of anything is because things havenâ€™t changed all that much over the last year â€“ barring a necessary shift towards low-voltage oriented ICs (~1.30V to ~1.50V) from the likes of Elpida and PSC. Parts of these types will eventually become the norm as memory controllers based on smaller and smaller process technology, like Intelâ€™s 32nm Gulftown, gain traction in the market. While voltage requirements have changed for the better, factors relating to important memory timings like CL and tRCD havenâ€™t seen an improvement; weâ€™re almost at the ...Â How did you know I was afraid to ask!! Reply. 0ldman79 - Tuesday, May 28, 2019 - link. Victims are asked to transfer funds to Bitcoin wallet 1Mz7153HMuxXTuR2R1t78mGSdzaAtNbBWX. A few hours after the attack began, the wallet was receiving transactions with the requested amountâ€"some victims preferred to pay the ransom without waiting for researchers to analyze the malware and come up with a file recovery tool. Within a few hours, the number of transactions tripled. Resilient and able to spread rapidly. Right when you think you saw the weirdest film ever, something like Woody Allen's Everything You Always Wanted to Know About Sex But Were Afraid to Ask comes along. An episodic adaptation of David Reuben's book of the same name, the film asks seven questions about sex that wind up puttingRight when you think you saw the weirdest film ever, something like Woody Allen's Everything You Always Wanted to Know About Sex But Were Afraid to Ask comes along. An episodic adaptation of David Reuben's book of the same name, the film asks seven questions about sex that wind up putting t...Â Due to the film being an anthology, it is perhaps best to approach this differently than many of my other reviews.