Technical Report 1: Software Design Guidelines

Douglas Kirk
Department of Computer and Information Sciences
University of Strathclyde
Glasgow

What is design?

Design is the process of deciding what form an item should take during its creation. It is a solution to the problem posed by the intersection of forces that occur in a given situation. In general an ideal design is one which seeks to find the optimal resolution of each force that affects the problem while simultaneously trying to achieve a balance across all forces. This should ensure that the highest quality solution is found for a given situation. Unfortunately this ideal is not often found in practice. A design can be affected by a large number of forces, each of which interacts with other forces in complex ways. This means that as one quality of a design is improved (say ergonomics) another is worsened by the same change (for example cost). Also our perception of the importance of different design qualities can change depending on the particular stake that we have in a system. For example a company might be primarily interested in the cost of a product while a user may perceive its usability to be more important. This makes the evaluation of design difficult because it depends upon your point of view how successful a design was. Another problem which can occur is that there might be a poor understanding of how to address a particular design force. Often such areas are labelled as creative areas of design and there is a tendency in such circumstances for designers to exercise their ego in the creation of a solution. This can work but often it creates a air of mystery around the design and we become unable to critique quite why it works.

Despite these problems design is something humans are generally good at. From the most basic of flint axes to the space shuttle mankind has mastered many forms of design. Much of what we see around us in our day to day lives has been designed and to a greater or lesser extent it achieves its purpose. Cars drive, chairs let us sit and buildings keep us warm. For the past thirty years we have been trying to provide a similar sense of control over the creation of software artefacts and to date we have had little in the way of success. Why should this be the case? At its simplest software design consists of identifying what a system must do and writing code which embodies the algorithms that are required to achieve the desired functionality. However in reality software is not developed in a vacuum. Many other forces impinge upon the creation of software, including cost, development time and reliability. It is difficult to address these forces as their affect on software is not well understood. One cannot look at a software design and say with confidence whether it can be built for a particular cost, timescale or reliability because there is little understanding of how a particular software structure affects those forces. What is the cost of an If statement? How much longer will this algorithm take to build than that one? How reliable will this class be? It is also difficult to set absolute values for forces and to prioritise them appropriately. How reliable must this program be? How efficient? Which is more important?

Software is intangible. It is non physical thought stuff and for a species that evolved in a physical world (and whose mental machinery has equipped for physical problem solving) it is hard to adjust to. Being free from physical restrictions also has another price to pay. With no physical costs to bog it down the scale and complexity of software gets elaborate very quickly. The complexity of modern software applications challenges the limits of human problem solving ability. There is simply too much software in a typical application to properly consider and evaluate all the implications of its design.

Software is not going to get any less complicated to design. Software developers have to face the challenge of designing and building increasingly complex systems. How will they cope? So far the problem has been addressed using two intertwined approaches. Increasing the level of abstraction of program entities and increasing the modularity of software. Both of these approaches can be seen as an attempt to localise design decisions into fewer more manageable chunks of code (and are borrowed from our success in other engineering disciplines). To some extent these mechanisms must work. Software has scaled from tens of lines to millions of lines of code over the past fifty years but such approaches rely heavily on the ability of the designer.

Abstraction and modularisation have led to the development of the object oriented paradigm. This is a powerful paradigm for the creation of software applications, but it is not perfect. One problem that many developers experience is the difficulty of deciding which aspects of a problem to encapsulate into classes. It turns out that decomposing a system into classes is an ill posed problem. For any one program there are an infinite number of possible decompositions and little if any guidance about which is best.

The same problem occurs in other aspects of the object oriented paradigm. When splitting behaviour into methods within a class, when creating inheritance hierarchies and when linking classes together into compositions there are a number of viable alternatives and no guarantee that a developer will pick the best option. Do we need to pick the best? Surely if object orientation is providing support for abstraction and modularisation then design will improve regardless. The actual choice itself can't matter than much, can it? Well it appears that it does. As long ago as 1972 Parnas (Parnas 1972) argued that modularisation alone was not enough. Instead decomposition had to take place along natural faults in the problem so that parts that changed together at a particular point in time were separated from the rest of the system allowing them to vary independently. Abstraction fares little better. In 1987 Liskov (Liskov 1988) claims that well designed inheritance hierarchies should exhibit a strict subtype relationship between parents and children to allow type substitution to take place. Whether or not you agree with their advice is not the issue, what is important is that both authors have argued for the creation of some types of class over others. To misquote George Orwell it appears that; "All classes are created equal but some classes are created more equal than others."

Parnas and Liskov are not the only people to offer such advice. Many other authors have proposed their own guidance about software design. Such rules or heuristics provide developers with some support to assist in the creation of better software designs but despite their availability design guidelines are seldom used. There are at least two reasons why guidelines have not been more widely adopted. Firstly, they are not widely publicised or taught so many developers remain ignorant of the available advice. Secondly, there is little evidence that the proposed guidelines actually improve software design. Developers are, understandably, reluctant to invest in guidance without demonstrable productivity or quality benefits.

This paper tackles the first problem, education. It presents a number of design guidelines from prominent sources in the literature and compares and contrasts them to identify their consensus of opinion and to highlight areas of controversy about software design. It also shows that the available guidelines are not uniformly distributed over the forces that affect software design but instead clustered around the concepts of modularity and abstraction skewing our perception of successful design.

Design Forces

A range of forces affect design. Before looking at the heuristics which aim to improve design quality it is useful to identify the type of forces that occur in software. These serve as a requirement list for design and draw attention to the wide range of factors that have to be considered when creating software. The following list is drawn from several software engineering textbooks ((Pressman 1994), (Meyer 1997), (Somerville 2001) and (Ghezzi et al. 2002)) and presents a representative list of the types of forces commonly associated with software.

- **Correctness**: This describes the ability of the software to meet the functional goals specified in its requirement document. A correct system is one that meets its specification.
- **Robustness**: Refers to the ability of a software system to react appropriately whenever an error condition occurs.
- **Flexibility**: This refers to the ease in which the software can be adapted during maintenance.
- **Reusability**: Describes the ability of the software or a part of the software to be reused as is within a different context.
- **Comprehension**: This quality describes how easy it is for developers to understand the intent and purpose of parts of the program
- **Efficiency**: This describes the concern over how much resource the software will consume. This will typically be measured in CPU time or storage space.
- **Portability**: The ability to transfer a program from one computer architecture to another. The easier this is the wider the potential market for the application.
- **Usability**: Refers to the end user experience. How intuitive is the software? What kind of learning curve do users go through? Usability addresses these types of questions.
- **Testability**: This describes how easy is it to check that the program is correct with respect to its specification.
- **Security**: How safe is the data contained in this application from outside attack? Does this software open up other vulnerabilities which might be exploited by a hacker? Security is the process of minimising these risks.
- **Cost**: A fundamental question of any design. What will it cost to build?
- **Time**: Another fundamental question. How long will it take to build?

This list of forces may not be complete others may exist which have not be included and the relative importance of each of these qualities will vary from project to project. Nevertheless this list serves to show at least some of the important qualities that designers have to be concerned with. Good design has to at least reflect each of these qualities, and try to find the optimum balance between them.

Design guidelines

This review looks at the work of three leading advocates on good object oriented style. It compares the work of Fowler and Beck (Fowler 2000), Riel (Riel 1996) and Meyer (Meyer 1997) to try to define a core set of properties that can be considered good design in object oriented programming. It will also consider which forces the various heuristics support and attempt to present a picture of the coverage of design forces provided by modern heuristics. This review is notable as much for what it leaves out as what it addresses. Many other works of design advice do exist, in fact that is hinted at in the introduction. Why have these works not been included? In part the answer is space, there are a great many to chose from and this review has selected the most comprehensive and solid collections of advice available. The other answer is that to a great extent the collections that have been selected draw upon much of this other work, either accidentally or intentionally and so these three authors between them capture much of the advice that is currently available.

Fowler and Beck

In his refactoring book Fowler co-authors a chapter with Kent Beck in which they describe a list of common design problems that can occur in object oriented programs. Their descriptions are presented from the perspective of problems with existing software and the intention is that once recognised the problems can be addressed by the refactorings presented in the remainder of the book. A total of twenty one design problems, they label them bad smells, are identified in the chapter. These address a heterogeneous collection of issues from the size of a class to the reuse of library classes with little apparent structure or rationale behind their selection. To ease their description this review groups smells into four categories, those relating to the identification of classes in a design, those relating to the communication of classes, those relating to the definition of the class interface and a miscellaneous group of smells which do not belong to any particular category.

The smells concerning the identification of classes within an object oriented design all relate to situations where classes should exist in the design but do not. Abstractions which should have been translated into separate classes have either been lumped together into bulky overweight classes or have been denigrated to attributes and details of other classes. The first smell in this category is called large class. This smell describes the situation where multiple abstractions have been forced into one class increasing its size. Fowler and Beck observe that large classes often contain a lot of duplicate code and can be large in two different dimensions; either by containing too many instance variables or containing too much code. The solution in either case is to look for collections of related variables and methods to divide into separate classes. They also mention that the way other classes in the system use the large class can provide clues about which sets of methods belong together (i.e. those that are called together may belong in the same class).

Another smell relating to class identification they call data clumps. This smell occurs wherever a group of data is used repeatedly throughout a system without being encapsulated within a class. Fowler and Beck claim that such proto-classes can often be found within the member variables of another class or in the parameter list of methods. They recommend a simple test for class cohesion. Whenever a data clump is identified the developer should consider deleting one of the data values in the clump. If the remaining items no longer make any sense as a class then, Fowler and Beck claim it is a sure sign that there is a class waiting to be detected.

Fowler and Beck also describes a smell relating to the reluctance of developers to model simple classes within their designs. They call this smell primitive obsession, because of designers' over-enthusiasm to model simple classes using primitive data types. They illustrates with some examples of simple abstractions that they believe should be modelled as classes, including money, telephone numbers and post codes. They argue that encapsulating these ideas behind a well defined interface protects them from future modification and improves the comprehension of the resulting code.

Bad smells in addition to identifying opportunities to add classes to a design also present advice about when to remove existing material. This includes the removal of methods and sometimes entire classes from the design to improve the cohesion of the remaining system. Speculative generality is a bad smell concerned with the removal of excess functionality from a class. This occurs where the interface of a class has grown to include methods which are not needed now but are included in anticipation of future needs. Fowler and Beck dislike such anticipation because it can often be wrong. For example, the additional method might never be used or an alternative unforeseen method might be required in its place. They point out that speculative generality can be detected when the only users of a method are its test cases. Fowler and Beck recommend removing speculative features as they argue that they make the class harder to understand and maintain.

Fowler and Beck also recommend the removal of data classes from a design. A data class is a class which has been created to hold data values for other classes in the system. When this occurs there is a lack of cohesion amongst the participating classes. The data class is not taking enough responsibility for its data and is not hiding its implementation sufficiently from the rest of the system. Other classes are exploiting this to include behaviour which ought to be beyond their capabilities. Fowler and Beck recommend moving the code that manipulates the data inside the data class and removing public access to the data. Closely related to the idea of a data class, in that it concerns a class talking insufficient responsibility within the system, is a lazy class. Fowler and Beck differentiate these from data classes because their implementation detail remains hidden, they are simply not doing enough work to justify their existence in the system. They observe that often such classes arise because changes to the system have removed responsibility from the class. Fowler and Beck recommend removing such classes by merging what little behaviour they contain into other parts of the system.

Fowler and Beck also identify a group of smells which appear to define their philosophy for the creation of cohesive classes. Stated informally this might be described as, code which changes together belongs together. Two smells, shotgun surgery and divergent change, present their thoughts on class cohesion. Both smells describe how the system responds to change. In shotgun surgery a modification causes alterations across several classes. Fowler and Beck observe that in such situations it can be difficult to find all the relevant places to make a change and easy to miss an important modification. In divergent change the situation is reversed and a single class is continually reacting to modifications for different aspects of the system. In both situations Fowler and Beck recommend arranging functionality across classes so that there is a direct correspondence between common changes and classes.

Fowler and Beck also observe that feature envy can occur when classes are not cohesive. Feature envy describes the situation where a method needs access to private details of another class. Often this can result in accessors being created in the public interface for the class to expose its implementation detail to the remote method. Fowler and Beck argue that this is seldom the correct solution and a more cohesive approach is to recognise that the envious method actually belongs inside the class and to move it so that it has direct access to the class's implementation. They point out that while this approach results in a more cohesive class sometimes the needs of an application actually demand that it be violated. In particular a number of design patterns (e.g. Strategy and Visitor), which are widely regarded as useful object oriented structures violate this principle.

The feature envy smell also helps to address class coupling. The decision to place data and methods together in a class is by default a decision not to create a communication between classes. Feature envy is essentially a transformation which removes a source of coupling between classes to improve their cohesion. Similarly Fowler and Beck describe another smell called inappropriate intimacy. This smell occurs when two classes exhibit a lot of coupling indicating that their implementation details are not appropriately hidden. As with feature envy the solution is to try to improve the cohesion of both classes. This may involve moving methods between each class or in some cases may require the creation of a new class, which the existing classes can share. Fowler and Beck also point out that this problem can be particularly common in inheritance hierarchies because of the close relationship which tends to exist between subclasses and their parents.

Fowler and Beck identify two bad smells related to composition in object-oriented systems, message chains and middle man. Message chains refer to the structure that can exist between communicating objects of a system. When one object wishes to obtain a reference to another and has to go through a number of intermediate objects to obtain it this chain of references is known as a message chain. Fowler and Beck consider this to be a bad smell because such chains couple the client to the structure of the navigation. This makes the code brittle to any change in the structure of the message chain. The solution they propose is to try to break the chain in two ways. Where possible the structure of the chain should be hidden from the client in a neighbouring class so that a client has no idea how the reference is calculated. Fowler and Beck comment that hiding the chain in this way can be overdone resulting in the creation of middle men (see below). So as an alternative they recommend moving the functionality of the client class further down the message chain to reduce the amount of communication required to reach the destination.

The middle man smell refers to an object which acts as an intermediary between two other communicating objects but which does not add significantly to the computation. Fowler and Beck argue that in such cases it is usually better to communicate directly with the object that knows what is going on. They suggests two ways to remove the middle man; either move code from the middle man into the client, or if there is additional behaviour being added by the middle man replace it with a subclass of the target object.

Fowler and Beck also provide a number of bad smells concerning the structuring of inheritance hierarchies. They suggest that switch statements are a bad smell because they lead to duplication within source code, parallel inheritance hierarchies should be avoided and subclasses should not refuse to accept the interface of their parent class. The argument against switch statements is that they make the system harder to modify as there are typically a number of switch statements spread throughout the system and alterations to one of them require changes to the rest. Fowler and Beck claim that in many cases the switch statement is used to modify a methods behaviour depending upon the type of a parameter. This they claim can be more effectively modelled using polymorphism, which allows different versions of a method to be executed at runtime depending upon the type of the object that the method is invoked from. Polymorphism doesn't lead to code duplication as the different behaviours are distributed across an inheritance hierarchy and the language runtime carries the responsibility of selecting the correct method to invoke. Fowler and Beck points out however, that polymorphism is expensive to set up and in circumstances where there are only a few alternative behaviours affecting a single method in the system then polymorphism can be overkill.

Another part of Fowler and Beck's change together stay together credo can be found in their advice about inheritance. In particular the bad smell concerning parallel inheritance hierarchies. Parallel hierarchies refer to situations where changes to one hierarchy must be reflected with symmetrical changes to the other. Fowler and Beck argue that such situations actually reveal a poorly formed hierarchy and they recommend merging the two related hierarchies into a single cohesive structure.

A final bad smell related to inheritance is called refused bequest. This smell exists whenever a subclass refuses, or doesn't need the implementation provided by the parent. Although Fowler and Beck identify this as a bad smell they admit to using it in their own code and do not consider it to be particularly harmful. They take a much firmer line when a subclass refuses to accept the interface of a parent. In this situation they recommend removing the subclass and replacing it with composition.

Fowler and Beck offer three pieces of advice concerning the construction of a class' interface. They recommend that methods in an interface ought to be short, parameter lists kept small and that consistent interfaces be used across similar classes. They claim that short methods make classes easier to understand and increases flexibility as they are easier to compose together. They identify sections of code which require a comment to be understood as potential areas to be replaced with a method. They also recommend looking for candidate methods around conditional statements and loops.

A second bad smell for a class interface is a long parameter list. Fowler and Beck argue that this is a hang-up from procedural programming where developers were taught to pass parameters in order to avoid global data. In the object oriented paradigm this situation changes because classes represent another level of state, between the global state of the system and the local state of a method. This they argue allows parameter lists to become much shorter, which brings benefits of comprehension, and maintainability to the code. The main way to reduce a parameter list's size is to pass objects rather than individual data items. Fowler and Beck also suggest using references from the local object's state to obtain other data values. They makes one exception to this rule they suggest that parameter lists should be used when trying to avoid a dependency between two data types in the system (by passing primitive data rather than the complex type) but they point out that the larger parameter sizes could easily outweigh the benefit of reduced coupling and the decision about which to use is subjective.

A final piece of advice about class interfaces comes from the smell; alternative classes with different interfaces. This smell describes the situation where two classes in the system do essentially the same thing but have a different interface signature. Fowler and Beck's advice is to create a uniform interface across such classes and to consider the potential to create an inheritance hierarchy to avoid duplicate code.

The final collection of smells presented by Fowler and Beck are not closely related to any of the other categories but include miscellaneous warnings about other areas to avoid in design. They argue that duplicated code is a smell that should be eliminated wherever it is detected. They suggest that in inheritance hierarchies this will result in common code moving up the hierarchy and in classes which are not related via inheritance the common code should be encapsulated into a new class which the existing classes can share via composition. Another smell they identify is called temporary field. This relates to the practice of storing temporary results in a variable. They do not have a problem with this per se but they do not like storing the temporary value as part of the state of the class. When this occurs they point out that the variable will at times contain uninitialised values and this means that the developer must understand much more detail about the class in order to use it correctly. Another bad smell that they object to has been named incomplete library class. This smell refers to the situation where a developer can find a reusable class which contains the majority of the code they need but not everything. This presents a dilemma for the re-user as it is seldom possible to alter third party code. Fowler and Beck recommend creating local wrappers to add extensions to the class. The final smell that Fowler and Beck describe is called simply comments. They argue that parts of a program which are extensively commented probably represent areas of bad design; the comments acting as a crude remedy for the confusion that the design causes its users. They recommend that where areas of code appear to be overly commented the code be carefully inspected to see if it can be refactored into a simpler more comprehensible structure.

Table 1: A summary of Fowler and Beck's design heuristics

Heuristics
Large class
Data clumps
Primitive obsession
Speculative generality
Data class
Lazy class
Shotgun surgery
Divergent change
Feature envy
Inappropriate intimacy
Message chain
Middle man
Switch statements
Parallel inheritance hierarchies
Long method
Long parameter list
Alternative classes with different interfaces
Duplicated code
Temporary field
Incomplete library class
Comments

Riel

In his book on object oriented design heuristics Riel presents a detailed list of heuristics available for the object oriented paradigm. His work is based partly on his experience as a consultant and teacher of object techniques but also builds upon the work of others notably Liskov, Lieberher and Holland (Lieberherr and Holland 1989), Johnson and Foote (Johnson and Foote 1988) and Wrifs-Brock et al. (Wrifs-Brock et al. 1990). His heuristics cover many aspects of object design but are mainly restricted to issues surrounding the modularity and flexibility of software. Riel's heuristics are organised by topic with each taking up a separate chapter in his book. He begins by discussing heuristics concerning classes and objects, and then describes common problems face by procedural developers when moving to the object paradigm. His third chapter offers advice on the relationships that can exist between classes within a system. The fourth and fifth chapters discuss heuristics regarding inherence and multiple inheritance, respectively and the final three chapters present miscellaneous advice concerning class associations and rules for optimisation of software.

Riel's discussion of classes and objects begins with a heuristic to keep all data hidden and a related heuristic to only access it via the public interface of the class. Riel argues that this restriction is essential to preserve the maintainability of software as it limits dependencies upon data allowing class implementations to change freely provided the interface remains stable. He later underlines this position with a further heuristic to avoid putting implementation details in a class's public interface and another heuristic which states that classes should only communicate via their public interfaces or not at all.

Riel's next heuristic is to minimise the number of methods within a class because he argues large classes are hard to understand and reuse. The size of the class interface making it difficult to understand what the class does and what functionality is available from its methods. This heuristic is followed by another which advises developers not to clutter the class interface with unnecessary details. Such advice seems redundant but underlines Riel's position to make class size as small as possible.

Riel advises that all classes in a system should share a minimal common interface. This he argues is a convenience mechanism which enables developers to have a starting point of methods from which to begin to understand the class. Whether this is useful or not for comprehension is debatable but it is a feature of many popular object oriented languages (e.g. Java, C#, Eiffel) and as such is already available for users of those languages.

Riel's final set of heuristics in the opening chapter concern the identification of good abstractions for classes in an object oriented design. He provides a heuristic which advises developers to create classes which model one and only one key abstraction. He defines a key abstraction as a main entity with a domain model although this definition appears both unhelpfully vague, how are we to recognise these key abstractions? And also too limiting; what about design patterns and implementation classes which are not represented by the domain model? Riel provides another heuristic which states: keep related data and behaviour in one place, and its converse heuristic, move unrelated data into another class. He argues that failure to observe these rules will damage encapsulation as details of a single abstraction will be spread over several classes. A final heuristic argues that classes in a system should model concepts and not the roles that an object might play. Riel provides an example of this situation where mother and father objects could be modelled by two separate classes (Mother and Father) or a single Parent class depending on the interface that is required on both classes. Riel points out that this problem is highly dependant upon the specific context of each system. In some cases the former will be the correct answer in other cases the latter will be preferred.

The second chapter of Riel's book presents some advice for procedural programmers converting to the object paradigm. Most of the heuristics in this section continue the theme of identifying good abstractions to convert into classes. He begins with several heuristics which aim to prevent the creation of god classes in a system. Riel observes that procedural developers often try to recreate the central thread of control prevalent in the procedural paradigm when they move to object orientation. Unfortunately this can result in the formation of god classes, large classes which end up performing the majority of the work in the system. Riel does not explicitly state why he disagrees with god classes but one can surmise that such large classes make comprehension harder and make the system less flexible to change. His advice takes the form of four heuristics. First he advises developers to distribute functionality as uniformly as possible across the top level classes of the design. The second heuristic is to avoid deliberately creating god classes in the system. He observes that class names such as Driver, Manager, System or Subsystem often indicate potential god classes. The third heuristic looks for the opposite of god classes, data classes, it advises developers to look for classes with a large number of accessor methods on their public interface. This implies that the class is exposing too much of its state to other classes and this might indicate the presence of a god class elsewhere in the system. His final advice is too look out for classes with too much non communicative behaviour, (methods that operate on a isolated subset of data items within a class). Riel's experience indicates that god classes often have such non communicative behaviour, although it is not clear how much non communicative behaviour is too much.

Riel next provides a heuristic which encourages developers to model the real world where possible. He argues that this is important from a maintenance perspective where mimicking the real world will help developers to understand the design better. He points out that this advice is qualified because there are a number of situations where modelling the real world is not advisable. He cites the distribution of system intelligence, avoidance of god classes and the desire to keep data and behaviour in one place as some potential motivations not to model the real world. It is worth noting that even without these exceptions the heuristic is of dubious value because it assumes that developers will broadly agree upon the subdivision of a problem into a set of classes. If this were really true then there would be little need for so many heuristics concerning the process of creating classes from requirements! A related heuristic which appears later in this chapter is to avoid modelling agent classes in the design. Agents are classes which derive from a literal translation of the problem from the real world. Riel provides an example of a Library system which might include a librarian as an agent class. Riel argues that if the librarian only exists to search for and retrieve books then it should probably be removed from the system as the behaviour it contains could be better modelled directly within the collection of books.

The remainder of the chapter presents advice for another problem that Riel claims affects procedural developers when moving to the object paradigm, the creation of too many classes. He calls this the proliferation of classes problem. His first heuristic is to remove irrelevant classes from a design. Irrelevant classes are classes which although they exist within the problem domain contain no real functionality. Riel argues that procedural developers are likely to misappropriate such candidates and convert them into real classes. He suggests in many cases these proto-classes can often be demoted to attributes of another class. A closely related heuristic is to eliminate classes which are outside the system. Once again these are classes created from the domain model but which do not have any meaningful behaviour in the system. Riel advises that such classes be removed from the system. Finally Riel counsels developers to avoid the mistake of turning operations into classes. He claims this is a leading cause of the proliferation of classes problem but can be detected by looking for classes which only contain one piece of meaningful behaviour. He also recommends treating classes whose names contain verbs as suspect. His solution to this situation is to remove the class and relocate the method to another class within the system.

Riel's third chapter deals with the relationships than can exist between classes. In general his advice is to have as few links as possible between classes but he also provides some heuristics governing the composition of classes and the definition of semantic constraints upon a class. He argues that the number of classes an object collaborates with should be minimised because intra class communication makes programs harder to understand. To effect this change Riel recommends looking for areas of the call graph of a program where one class communicates with several others and attempt to encapsulate the communication in a larger class which contains the group. He suggests that making this modification pushes the communication into the implementation detail of the containing class, making it easier to modify in the future. Similarly he advises that the number of messages sent between collaborating classes, the types of messages sent and the fanout of a class (a product of the number of methods and the messages they send) should all be minimised. This is essentially an argument for the creation of loosely coupled classes. Riel observes that each additional level of communication in a system, while not as significant as the original communication, increases the complexity of the program and should be reduced where possible.

Riel provides a number of guidelines to assist in the creation of containment relationships between classes. A containment relation occurs whenever a class contains a reference to a class which is not a value type. Riel's first piece of advice for containment is that the containing class should be the only class which communicates with the contained class. He points out that classes which violate this principle (with the exception of dedicated container classes) are breaking the heuristic to keep related data and behaviour together. Later he proposes a related heuristic which captures the other side of the relationship, namely that although a container knows about its contents; the contents should not know anything about the container. However, Riel points out that in some cases it is desirable to have a contained object know about its container if it will significantly reduce the amount of class communication in the system. Riel also suggests that a practical limit should be placed on the number of classes contained by a class based on the limitation of short term memory. Such advice seems dangerous because as Miller points out (Miller 1957), the technique of chunking data can have a significant effect on the number of items that can be managed effectively in short term memory. In addition, imposing a limit of the size of a class based on cogitative limitations might easily fragment an abstraction across several classes something Riel is also keen to avoid. Riel calls for containment relationships to be structured into narrow and deep containment hierarchies. This advice is partly driven from his belief that classes should contain few objects (hence a narrow hierarchy) and partly from a desire to create reusable software. Riel argues that by creating deep containment relationships rather than broad shallow ones developers maximise the number of entities available for reuse. Although it could be argued that this approach is guilty of creating a large number of relatively trivial classes which would seem to violate his proliferation of classes principle. His final piece of advice concerning containment is to prevent communication between objects in the same level of a containment hierarchy. Riel objects to this because he believes it makes the individual objects harder to reuse and increases the complexity of the system. Instead he suggests that the containing object should take responsibility for passing messages between its contents.

Somewhat strangely, given the context of the chapter, Riel goes on to define a heuristic for class cohesion. He argues that most of the methods should be using most of the data members most of the time. This is a data driven definition of cohesion carried over from procedural programming which some authors have criticised as being irrelevant for the object paradigm (Ott and Bieman 1998). Riel believes that failure to comply with this advice leads to non communicative behaviour which effectively denotes a relationship between two classes which have become fused into one large class.

Riel ends the chapter with a number of heuristics concerning the modelling of semantic constrains in classes. A semantic constraint reflects a state or set of states which a class is not allowed to get into. Riel provides an example of a car class which can only accept certain combinations of car type and engine. Where possible Riel recommends enforcing these constrains in the definition of the class itself. This provides the strongest level of protection but Riel observes that it can often lead to a proliferation of classes (one for each allowable state). To overcome this he proposes an alterative heuristic which is to place the constraint in the class constructor, this allows the class to evaluate whether a particular combination of objects can be fused together into an object or whether they violate a constraint can cannot. Riel advises that the constraints be placed as deep in the containment hierarchy as possible to limit the scope of the constraint as much as possible. Finally he observes that some semantic constraints are volatile while others are fairly stable and recommends different approaches to modelling each situation within software. Where constraints are volatile he recommends placing the constraint in a separate object (a look up table) which the constrained class can consult to find out if a particular combination of parts is valid. If the constraints are more stable this extra class can be omitted and the table of options can be modelled directly within the constrained class.

Riel next turns his attention to inheritance and provides a number of guidelines for the use of inheritance and multiple inheritance in software designs. He observes that inheritance is used for two purposes within a design. It can serve to generalise a range of ideas into a single concept or it can be used to specialise a general concept into a more specific one. Riel suggests that both applications of inheritance are important but that generalisation typically occurs earlier the in design process and is the harder of the two to get correct. He also points out that many developers confuse inheritance with containment. This leads to his first set of heuristics in this chapter which aim to define inheritance and the situations in which it is applicable. His first heuristic argues that inheritance should only be used to model specialisation. Riel argues that inheritance should be used in this way because it is a white box technique which matches the complexity of the specialisation relationship. In effect what he is saying is that with specialisation you have to understand the behaviour of the parent class to make an effective modification. Inheritance provides such insight because the subtype gets access to details of its parent class. Riel argues that this close mapping between the power provided by the inheritance mechanism and the requirements of specialisation make the two perfectly suited for each other. Riel later reinforces this position with a heuristic which advises developers not to allow null overrides in subclasses. Riel's argument is that if null overrides are allowed then it is impossible to enforce the specialisation relationship between parent and subclasses – in effect anything can be a subclass of anything else. This argument has some precedence other authors argue similar things (e.g. (Liskov 1988), (Fowler 2000)) but it is not a universally held position and as we shall see later Meyer (Meyer 1997) argues against this point of view claiming that no hierarchies are perfect and some exceptions have to be allowed via null overriding.

Riel's next heuristic concerns the depth of inheritance hierarchies. He suggests that they should be deep as this provides a fine grained decomposition of classes for reuse but he observes that in practice hierarchies can become large and unwieldy. He recommends that for ease of comprehension hierarchies should be limited in depth to the number of items a person can fit into short term memory. He also advises developers to factor common data or behaviour as high into the hierarchy as possible. This is derived from a reuse argument where shared code should be as high as possible to prevent code duplication but Riel also points out that it has benefits for the consumers of a hierarchy. He observes that if base classes are suitably enriched with functionality it can enable their users to operate with a more general type, which can in turn make that method applicable to a wider audience.

Riel warns developers not to model dynamic relationships with inheritance. Such implementations will toggle types at runtime as the dynamic state of the object changes. Riel points out that this is both inefficient and confusing, it can also lead to a proliferation of classes as each state is being modelled with a separate type. Instead Riel recommends using composition whenever relationships are liable to change dynamically. This allows object to be associated with a variety of other objects as their state changes. Similarly Riel provides a heuristic to avoid creating hierarchies of objects. He claims this problem can occur when a developer sees the instances of a class as if they are distinct types. He points out that modelling these concepts as classes leads to a proliferation of classes as each class will typically only produce one instance in the program. He expands upon this topic with another heuristic which argues that if a developer is trying to create new classes at runtime then this almost always is a mistake and the dynamic classes are in fact objects which could be modelled as instances of a static class. Riel also warns developers to avoid modelling optional containment with inheritance. This is a continuation of his argument against dynamic relationships because the hierarchy is not stable and likely to grow dramatically if new options are added. Riel provides an example of a house which might optionally contain heating, lighting, plumbing systems and so on. Each combination of systems has to be allocated its own class in case a user wants that particular combination of components. This is another example of Riel's proliferation of classes problem because every new option added to the hierarchy will result in many new classes to handle all the possible combinations of components. Riel recommends that such situations are modelled using composition rather than inheritance.

Riel advises that hierarchies should be created and modified with their global structure in mind rather than simply thinking about the more obvious parent child relationship. He provides a heuristic which states that developers ought to construct reusable frameworks rather than reusable components. His argument is that a well thought out framework will help developers when making specialisations, as there will be a series of well thought out locations for them to specialise from. However he seems to have gone off topic because it is clear that by framework he is talking about the generic design for a range of applications. Such structures cannot be modelled using inheritance alone. They require a combination of both inheritance and composition to be effective. He also assumes that the additional complexity of such designs is always a good thing. In some circumstances a framework design could easily be overkill and not provide benefits commensurate with its cost.

In addition to generic advice bout the shape of inheritance hierarchies Riel provides a number of heuristics about the more detailed relationships between a parent and its children. His first heuristic is that derived classes should know about their parents but not vice versa. This mirrors his advice about containment earlier in the book and reflects the goal of keeping elements as decoupled from each other as possible. In this case making it easier to restructure a hierarchy because lower nodes are less dependant upon upper levels. Another heuristic which is related to this goal is his advice to keep all data in a base class private (as opposed to being protected). Once again this helps reduce the coupling between layers in the hierarchy making them easier to reorder.

Riel advises that all abstract classes must be base classes and also that all base classes must be abstract. The first of these rules is easily explained because an abstract class which is not sub-classed can never be used. However the second is more confusing. Riel is arguing that an abstract root will make a hierarchy easier to change as additional implementations can be easily added without modifying the root but this implies that the abstract class is itself stable and unlikely to change which is impossible to guarantee. Riel suggests that the main benefit of this approach is to prevent global renaming if the root of the hierarchy changes but this argument is undermined by modern refactoring tools which make such renaming operations easy to perform.

Riel also provides a number of heuristics to help pigeonhole the type of design to model. He suggests that if two classes only share data between each other then inheritance is not appropriate and the shared data should be encapsulated into its own class which the other classes share via a composition relationship. If on the other hand the classes share data and behaviour then they are suitable for inheritance and a common base class should be extracted from both classes which they can inherit from. Finally if two classes share a common interface they should only be merged into an inheritance hierarchy if they can be used interchangeably as part of a polymorphic substitution.

Riel also argues against the use of explicit case analysis in object oriented software. He recommends using the polymorphism mechanism to differentiate behaviour based on type. This has the advantage over explicit type analysis that new implementations of the polymorphic method can be created without making modifications to the rest of the system.

Riel provides two heuristics for multiple inheritance. The first is to assume that it is a mistake and try not to use it the second is to prevent accidentally inheriting from multiple base classes. This first piece of advice would appear to invalidate the second as if it is not to be used then why bother placing further restrictions upon it. Riel is being droll with this heuristic, what he really means is that multiple inheritance is difficult to use correctly so if you don’t need it, don’t use it. Such advice is practical if somewhat unhelpful because it does not help to diagnose those circumstances where its use is frivolous from those where it is necessary. His second heuristic to avoid inheriting from multiple bases is driven by the fact that those bases may share a relationship which will prevent conformance operating correctly during polymorphism. Riel provides an example of an orange, which inherits from two bases, food and citrus fruit. He argues that this is dangerous because it fails to capture that fact that citrus fruit is itself a type of food. Therefore a circumstance could exist where an orange represented as a citrus fruit was forbidden being passed as a type of food despite the fact that it implemented the food interface. One speculates that this heuristic is seldom required as the particular set of circumstances that must exist to cause the problem seem somewhat contrived. Riel ends his discussion of inheritance by providing two questions which he claims can help to disambiguate between whether to use inheritance or composition in a design. Developers should ask; is this class a special type of the thing it might inherit from? Or is the thing from which the class might inherit actually a part of the class? Riel suggests that if the first question is answers positively and the second negatively then inheritance should be used while the opposite answers imply composition should be used.

Riel ends his collection of heuristics with a collection of miscellaneous advice which didn't make it into any of the preceding chapters. These include a heuristic to favour composition over association relationships because it keeps data and behaviour together. In also includes a heuristic to avoid the use of global data to perform bookkeeping operations on the objects of a class Riel argues that class variables or methods should be used instead. Two final heuristics occur in a discussion about code optimisation. He advises that physical design criteria should not be allowed to corrupt logical designs and in a reiteration of a previous heuristic advises; that the state of an object should not be changed except through its public interface.

Table 2: A summary of Riel's design heuristics[1]

Heuristic
Keep data hidden
Minimise class size
Common interface
Model key abstractions
Avoid god classes
Model the real world where possible
Proliferation of classes
Minimise class collaboration
Containment should be uni-directional
Containment should be deep and narrow
Classes should be data cohesive
Classes should enforce semantic constraints
Inheritance should only model specialisation
Inheritance hierarchies should be deep
Dynamic relationships modelled with composition
Create hierarchies as reusable frameworks
Subclasses must be decoupled from parents
Don't use explicit case analysis
Avoid multiple inheritance
Don't use global variables
Don't corrupt logical designs with physical constraints

Meyer

Bertrand Meyer provides his opinion about what is considered good object oriented design in his book on object oriented construction techniques. The core of the book teaches the object oriented paradigm using a straw man approach where the weaknesses of other paradigms are contrasted against the strengths of object orientation. In addition to teaching the basic principles Meyer intermixes some of his own experience about what structures makes effective classes and create good designs. Meyer's advice differs from his peers in its breadth of coverage; other authors

[1] Riel's heuristics feature a certain amount of repetition and overlap. For brevity this summary does not list each heuristic in his book but instead lists the central themes that he addresses.

focus on design as a means to promote flexibility and reusability of the underlying code. The importance of these forces is strongly argued by Meyer but he also addresses other issues, such as the need for correctness and robustness in the design of software. This makes Meyer's account one of the most complete discussions on software design currently available.

Meyer has structured his book into sections which combine together in a logical sequence. The first section presents the forces that software development must address, the second presents motivating arguments for object orientation, the third introduces the underlying concepts and the fourth teaches how to apply those concepts well. The remaining sections of the book deal with details such as language issues, development environments and tool support. Much of the book is concerned with teaching basic concepts, such as modelling using classes, creating methods and attributes etc which are of little interest to this discussion. Of greater relevance are Meyer's opinions about how to employ these elements correctly. His comments are scattered throughout the book but offer a range of design advice for object oriented programmers.

Modularity and reuse are important to Meyer and each receives its own chapter in his book. He argues that modularity is important because it promotes the extendibility and reusability of the resulting software. He defines the term using five criteria claiming that effective modules; should be decomposable into smaller pieces, composable (or re-composable) into larger systems, understandable in isolation from each other, continuous, in the sense that small changes to the requirements are localised within a module, and protected, where an abnormal condition in one module will be contained at that location and not spread to other modules. To ensure that modular systems present these properties Meyer also defines five rules and five principles which he claims will help to produce modular systems; The rules are direct mapping, few interfaces, small interfaces, explicit interfaces, information hiding. The principles are; linguistic modular units, self documentation, uniform access, open closed and the single choice principle.

Direct mapping refers to the relationship between the class structure of a program and the corresponding domain which it models. Meyer argues that it is important to maintain a clear relationship between artefacts in the problem domain and their manifestation in software. This eases the comprehension of the software and can help to produce a decomposition which will remain stable over the lifetime of the software. This advice appears to echo other authors who argue 'model the real world wherever possible'. However, Meyer is less enthusiastic, he believes that direct mapping relates not to a correspondence between the real world and the software model but instead captures the relationship between a model of a process (a model of reality) and the more specific software model that implements it. As Meyer points out the separation between the real world and the software world is often untenable. He asks what is real about a program that performs mathematical computations, or how to classify a compiler (a program which creates other programs)?

The few interfaces rule refers to the amount of communication that is allowed between classes in software. Meyer argues that classes which are too heavily connected to their neighbours will run the risk of propagating changes or errors to them. This connectivity also has an effect on the comprehension of a class and its reusability across contexts. Interestingly Meyer shows that the minimum number of links between classes can be produced through a centralised structure where all communication goes through a central 'master module' but he does not advocate this approach. Instead he argues that a more even distribution of interaction across the system is desirable as it creates a robust, extensible architecture.

The small interface rule is related to the previous one but is concerned with reducing the amount of information that is exchanged rather than the number of links between classes. Meyer argues that such communication should exchange as little data as possible because this will protect the code from changes to that data and prevent error conditions propagating between classes.

The explicit interfaces rule continues the advice on class interfaces by insisting that all communication between classes be public and clearly visible. In the object paradigm this essentially limits communication between classes to elements which are defined on the classes' public interface. Meyer argues that this rule is important for both decomposability and composition and helps to reduce the impact of changes and improve comprehension.

The final rule of modularity, information hiding, is an extension of explicit interfaces which forbids access to any part of a class except to that declared in its public interface. This allows a developer to hide implementation details behind the public interface. Meyer argues that this helps to reduce the impact of changes to a program. The implementation of a class can change dramatically without affecting any of the other classes which can only use its stable public interface.

The linguistic modular units principle states that the modularity of software should follow the syntactic units available in the chosen programming language. So in an object orientated language the units of modularity are classes and methods, where as in a procedural language it would be modules and procedures. This principle has little relevance to this work as we are already restricted to object oriented design and are more interested in how to define good modules rather than what structures are used to describe them.

The self documentation principle argues that each class should contain all information that is relevant to its comprehension. Meyer points out that the traditional approach of having separate code and documentation inevitably leads to synchronisation problems and it is better if code and documentation can be collated together into a single location.

The uniform access principle implies that all communication with a class should be via method invocation. This helps hide implementation detail as a developer cannot tell if a method is returning a pre-computed value or performing a calculation before returning the result. In his description Meyer actually goes further proposing that object oriented languages should provide syntactic support to blur the distinction between variable access and method calls but from a practical perspective few languages support this feature and his advice is perhaps best interpreted as to restrict access to a class' methods.

The open closed principle states that classes should be open to modification but closed to change. Meyer points out that these apparently contradicting aims are catered for in the object oriented paradigm via the use of stable interfaces and inheritance. A stable public interface on a class allows other classes to use its functionality with confidence that the interface is unlikely to change in the future. Inheritance allows variations of the interface to exist providing functionality appropriate to different circumstances. This principle minimises the impact of change and creates more flexible and reusable abstractions (as a number of variations can exist). The open closed principle also has a positive affect on program comprehension because it reduces the number of types the developer needs to maintain in their short term memory (as the majority of the code will refer to the stable interface rather than specific implementations).

Single choice is the final modularity principle described by Meyer. He describes it as a consequence of the open closed and information hiding rules. The single choice principle essentially implies that only one class in a program should have knowledge of the polymorphic variations that can be swapped in for a given interface. Meyer argues that this principle helps support the flexibility of software as whenever a new type is added to a hierarchy at most one location in the program must be changed.

In comparison to modularity Meyer's chapter on reuse doesn't produce much in the way of concrete advice. This does not suggest that Meyer considers reuse unimportant; on the contrary he argues that it is central to the production of timely, reliable and efficient software. But he also recognises that reusable software is difficult to create and claims that developers must first reuse others software before they can gain an understanding of how to design reusable versions of their own software. Meyer also argues that reusable software must be easy to understand and in particular believes that differences in the semantics of a program should be reflected in visible differences in the program text. He refers specifically to method overloading claiming that this practice is detrimental to program comprehension because it does not differentiate the behaviour of similar methods. Meyer would much rather have unique names for each method that perhaps allows a reader to infer that a set of methods are related but at the same time have some idea of how each differs from the other.

Meyer provides some support for design forces such as program correctness and reliability through the use of assertions. He argues that class structures by themselves are not enough to describe software there is often an additional need to explain conditions and invariants which are expected to hold during the execution of the software. Meyer proposes the use of pre and post conditions to define properties expected before and after a method call. He also recommends the use of class invariants which express properties that must always hold within a class. Meyer claims that adding assertions to a language increases its correctness, reliability and comprehension. However, he provides little advice about how to recognise where to apply assertions to code and also points out that assertion checking can have a significant performance impact.

In the later portion of the book Meyer specifically focuses on how to apply the object oriented paradigm well. He begins with the process of identifying classes within a design. He points out that this is a difficult task and making the correct decisions takes talent, experience and a certain amount of luck! He provides a variety of advice about how to recognise candidate classes but warns that no amount of guidance will make the task easy.

He warns developers to avoid the temptation to create useless classes. In Meyer's opinion useless class typically derive from an attempt to model the problem domain too literally. This can result in classes which sound like reasonable abstractions but actually contain too little relevant behaviour to justify their existence (this concept is similar to what other authors describe as a lazy class). Meyer doesn't provide any advice on how much functionality a class requires before it can be considered useful. Conversely Meyer also points out that it is easy to overlook important classes during analysis. He cites the ambiguity of typical specifications and the tendency to create solutions from scratch rather than build upon existing ideas. In short Meyer sums up class discovery as a tension between class suggestion and class rejection. This implies that a designer must iteratively evaluate candidate classes, looking for new alternatives and removing weaker candidates.

Meyer claims that a common mistake made by developers new to the object oriented paradigm is to identify and create classes which model procedural abstractions rather than domain objects. Meyer argues that such classes are often revealed because they perform a single task; have an imperative name or contain only one method in their public interface. Meyer's opinion is that classes are not supposed to do only one thing. Instead they should offer a range of services to the system. This advice seems to contradict Meyer's earlier advice to minimise the amount of coupling between classes (e.g. the few interfaces and small interfaces rules) but this is not the case. Those rules emphasize that communication with other classes must be as small as possible, but classes are still free to contain a large number of methods each of which take part in different communications or which are all related but absolutely essential to the task at hand. In either case it is clear that Meyer is not calling for a shorting of class size per se but rather an optimisation of is communication with its neighbours. Single method classes do not follow from either of those rules.

Another mistake Meyer has observed is a rush to classify classes into hierarchies before their interfaces have stabilised. He believes this to be a problem because classes will move about excessively within hierarchies and between hierarchies as their interfaces change. He would rather developers defined the interfaces then worried about allocating classes into hierarchies.

Meyer has also observed that mixed abstractions can malform classes during design. He argues that merging multiple concepts together makes the class harder to understand and that good design should seek to create classes which relate to a single well defined abstraction. As appealing as this notion is, Meyer does not provide any detail about what criteria may be used to help identify and separate related material from a mixed abstraction.

Meyer believes that well designed classes should have two types of method in their public interface commands and queries. Commands are methods with change the state of an object while queries return a value based on the state of an object. The roles are mutually exclusive preventing a method from becoming a command and a query at the same time. Meyer argues that the advantage of this separation is improved comprehension; its helps prevent methods getting too complex (as they have to be partitioned into commands and queries) and it provides referential transparency (the ability to substitute a method for a constant value) which makes the behaviour of the software more predicable. The division of a class interface into commands and queries is novel; other developers have not suggested similar separations and it would appear to have wide reaching affects on the interfaces that are produced. For example Meyer shows how the operation pop for a stack cannot be allowed because it both changes state and returns a value at the same time; instead he insists that a stack should have two methods, one to return the value at the top of the stack and the other to remove it. Meyer notes that a potential error developers can make is to create a class with no commands. Such a class is merely a data holder and is analogous to the data class problem identified by other authors.

Meyer also argues that parameter sizes affect the comprehension and usability of classes. In particular the larger the number of parameters in a method the harder it is to understand and use. He points out this is particularly important in an environment where classes are expected to be reused as this increases the likelihood that a method will be used incorrectly or not reused at all. He recommends reducing the size of parameter lists where possible. Meyer observes that parameters can be passed into a method for two distinct purposes, either to provide data for the method to act upon or to indicate a mode of operation. Meyer calls data parameters operands and mode parameters options. He argues that only operands are essential to the execution of a methods and recommends reducing parameter sizes by removing options. He points out that in many cases options can be represented by additional methods which set the class into a given mode. It is then possible to initialise the class into a default mode when it is created and then make calls to change the mode as required.

Unlike some authors Meyer does not see the size of a class as a particular problem, he reports anecdotal evidence of developers operating successfully with classes containing eighty methods. Rather than size being the problem Meyer believes that the issue is cohesion. He argues that his developers could cope with large classes because they contained one cohesive concept. This he argues, reduces the amount of information developers need to keep in their short term memory, i.e. they can think of a class by its general concept, rather than remembering that 'this is a class which does this and has methods for that' etc. When creating classes Meyer suggests asking four questions to determine whether a method should be placed within a given class. Is the feature relevant to the data abstraction represented by the class? Is it compatible with other features of the class? Is its purpose unique from all existing features? Does it maintain the invariant of the class? If these questions are answered positively then the method belongs in the class and should be added to it.

Meyer also provides some guidance for the creation of effective inheritance structures within code. He observes that developers often get composition confused with inheritance and proposes the same 'has a' and 'is a' rules that other authors recommend. However he does point out that these rules are not absolutes and care must be taken particularly with 'is a' to make sure that it is not actually an example of composition. He proposes the rule of change to help disambiguate these difficult cases. This rule states that composition should be used in place of inheritance whenever the object components may have to be changed at runtime. In addition he proposes the polymorphism rule which states that inheritance is more appropriate wherever entities if the more general type might need to become attached to the more specialised type.

Meyer describes another problem with inheritance hierarchies which he calls taxonomania. This condition occurs when developers have overly decomposed the levels of their hierarchy, producing a large number of small classes which are not useful accept as a location to inherit from. Meyer argues that each stage of the inheritance hierarchy should make sense as an abstraction by itself. This limits the potential to subdivide a hierarchy resulting in fewer and more useful classes.

Meyer proposes a number of different forms of inheritance (subtype, view, restriction, extension, functional, type, reification, structure, implementation and facility) and claims that any potential situation which falls outside those categories should not be modelled with inheritance. Similarly he also argues that within a single hierarchy the type of inheritance used should remain consistent. Meyer's most controversial advice about inheritance concerns overriding methods. Most authors argue that overriding to remove behaviour should be avoided as it risks damaging type substitutability. Meyer on the other hand welcomes null overriding claiming that it is an important facility because no hierarchy (in nature or software) is perfect and a mechanism has to be available to include exceptions within a hierarchy. Despite this difference in opinion with other writers it seems that Meyer still shares their disapproval of misusing null overriding. He argues that such exceptions are rare and a hierarchy with many exceptions probably isn't a hierarchy at all.

Table 3: A summary of Meyer's design heuristics

Heuristic
Direct mapping
Few interfaces
Small interfaces
Explicit interfaces
Information hiding
Self documentation principle
Uniform access principle
Open closed principle
Single choice principle
No syntactic polymorphism
Use assertions
Avoid useless classes
Don't model operations as classes
Don't mix abstractions
Command and queries principle
Reduce parameter size
Class cohesion important not size
Inheritance is-a rule
Composition has-a rule
Avoid taxonomania
Only use defined forms of inheritance
One type of inheritance per hierarchy
Null overriding should be allowed

Critique of heuristics

The list of design heuristics produced by Fowler and Beck, Riel and Meyer represent a considerable amount of knowledge about object orientated design. In the majority of cases each author has produced a unique set of design advice but there are a few areas where the same or similar advice appears across authors and also some incidents where there is a direct disagreement about the type of advice to follow. This section will discuss these areas in more detail.

Areas of agreement

The largest area of agreement centres on the problem of identifying an appropriate subdivision of a problem into classes. Fowler and Beck identify a number of smells related to this problem, large class, data class and inappropriate intimacy. While Riel provides a group of heuristics related to the problem of detecting god classes (essentially a misappropriation of functionality into one large class). Meyer agrees with both of these positions warning against mixing abstractions within a single class.

A similar problem is the creation of useless classes and again there is strong agreement amongst the different authors. Fowler and Beck warn against creating lazy classes, Riel describes the problem as a proliferation of classes and Meyer complains about the modelling of useless classes and the modelling of operations as classes. In each case there is an agreement that there is a minimum threshold of functionality that a class must cross before it can be considered a viable abstraction in a program.

Another area of agreement concerns the replacement of switch statements with polymorphism. Fowler and Beck describe the switch statement smell, while Riel advises developers to avoid explicit case analysis and Meyer more generally provides the single choice principle which argues that only one location in a program should contain knowledge about the types in a hierarchy.

Both Fowler and Beck, and Riel provide guidelines which encourage the developer to place data and behaviour together in a class. Fowler and Beck advising that what changes together should stay together, while Riel talks about modelling key abstractions and keeping related data and behaviour in one place. More generally both Riel and Meyer recommend creating cohesive classes an essentially similar argument to keeping related information together.

Meyer and Riel also agree upon minimising the number of collaborations that occur between classes; the need to keep data hidden in a class; and that dynamic relationships should be modelled via composition rather than inheritance.

Areas of disagreement

There are comparatively few areas of disagreement amongst the heuristics provided by different authors. Those that were detected were all differences of opinion between Riel and Meyer although this is probably a result of their similar forward engineering perspectives which are different to Fowler and Beck’s reverse engineering stance.

The first area of disagreement concerns whether to model the real world in object designs. Riel argues that this is desirable where possible because it helps developers to become familiar with the structure of the software. Meyer argues that there is no such thing as the real world but instead there is a need for direct mapping between the model of a given domain and the corresponding source code. To some extent this can be seen as a rather trivial distinction but Meyer’s separation from the physical world appears to be a clearer definition.

Riel and Meyer also disagree on the topic of class size. Riel argues that class size should be minimised because large classes are difficult to understand. Meyer directly contradicts this stating that large classes are not a problem provided they remain cohesive abstractions.

The final point of disagreement concerns the use of inheritance. Meyer argues that any type of inheritance should be allowed and that in particular null overriding is an important, although seldom used aspect of inheritance. Riel takes a much more restricted view claiming that only specialisation inheritance should be allowed and that to enforce this null overriding should not be allowed.

Comparing heuristics to forces

The following table presents a refinement of the design heuristics presented in this report. It collates them together using the previous discussion to merge related topics into one heuristic. Areas of disagreement are also shown. This table also presents information about which heuristics provide support for which design forces. From this it is possible to see which forces are well supported (and which are not) by current design advice.

Table 4: Design heuristics versus design forces

Heuristic	Correctness	Robustness	Flexibility	Reusability	Comprehension	Efficiency	Portability	Usability	Testability	Security	Cost	Time
Don’t mix abstractions	✘	✘	✓	✓	✓	✘	✘	✘	✘	✘	✘	✘
Model related data and behaviour in one place	✘	✘	✓	✓	✓	✓	✘	✘	✘	✘	✘	✘
Avoid lazy classes	✘	✘	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘
Don’t use explicit case analysis	✘	✓	✓	✓	✘	✘	✘	✘	✘	✘	✘	✘
Avoid long parameter lists	✘	✘	✓	✓	✓	✓	✘	✘	✘	✘	✘	✘
Minimise class collaborations	✘	✘	✓	✓	✓	✓	✘	✘	✘	✘	✘	✘
Information hiding	✘	✓	✓	✓	✓	✘	✘	✘	✘	✘	✘	✘
Rule of change	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘	✘	✘
Data clumps	✘	✘	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘
Primitive obsession	✘	✘	✘	✘	✓	✘	✘	✘	✘	✘	✘	✘
Speculative generality	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘	✘	✘
Feature envy	✘	✘	✓	✓	✓	✓	✘	✘	✘	✘	✘	✘
Message chain	✘	✘	✓	✘	✓	✘	✘	✘	✘	✘	✘	✘
Middle man	✘	✘	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘
Parallel inheritance hierarchies	✘	✘	✓	✓	✓	✓	✘	✘	✘	✘	✘	✘
Long method	✘	✓	✓	✓	✓	✘	✘	✘	✘	✘	✘	✘
Alternative classes with different interfaces	✘	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘	✘
Duplicate code	✘	✓	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘
Temporary field	✘	✓	✘	✘	✓	✘	✘	✘	✘	✘	✘	✘
Incomplete library class	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘	✘	✘
Comments	✘	✘	✘	✘	✓	✘	✘	✘	✘	✘	✘	✘
Common interface	✘	✘	✘	✘	✓	✘	✘	✘	✘	✘	✘	✘
Containment should be unidirectional	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘	✘	✘
Containment deep and narrow	✘	✘	✘	✓	✓	✘	✘	✘	✘	✘	✘	✘
Classes should	✘	✓	✘	✓	✘	✘	✘	✘	✘	✘	✘	✘

enforce semantic constraints												
Inheritance hierarchies should be deep	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
Create inheritance hierarchies as reusable frameworks	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
Subclasses should be decoupled from their parents	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
Avoid multiple inheritance	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Don’t use global variables	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Don’t corrupt design with physical constraints	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
Self documentation principle	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Uniform access principle	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
Open closed principle	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
No syntactic polymorphism	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Use assertions	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Command and query principle	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Avoid taxonomania	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗
Only use one type of inheritance per hierarchy	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Model real world / direct mapping	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Minimise size / size is unimportant	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Only model specialisation inheritance / use any recognised form of inheritance	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗

The above table shows that much of the available design advice is clustered around the topics of flexibility, reuse and comprehension. Some small amount of advice stretches to address efficiency and robustness but few go further. In particular there is a dearth of advice for topics such as portability, usability, testability, cost and time to market.

This distribution of design advice is troubling because it suggests that these areas of design are being underrepresented. To some extent the cause of this problem belongs with the traditional argument that software ought to be modular and abstract. This is a popular belief because software requirements are not set in stone and the design has to be able to accommodate change. Trouble is that modularity and abstraction decisions are not absolute. There are many alternative decompositions and the correct choice largely depends upon how the software will be expected to change in the future something which is difficult to predict. This places a greater significance upon flexible structures in design because the modular decomposition alone might not offer enough flexibility. These alterations drive up the complexity of a design which increases its cost and the time it takes to build. Reuse is another force which drives up complexity because it stresses the need to capture well defined interfaces and a granularity of functionality which will be suitable for a wide range of circumstances. This also drives up the cost of the software (although to some extent this may be amortised over a number of subsequent reuses). The emphasis on creating complex software suggests that it will also become more complicated to understand. It is interesting to note that a lot of the design advice presented in this paper aims to make software easier to understand. It might be interesting to measure how effectively such advice actually improves comprehension.

However, the previous paragraph comes dangerously close to suggesting that software should not be made as complicated as it is. While there are a number of advocates of this position, most notably the agile development community (Beck 2000), it would be wrong to insist on simplicity for its own sake. Instead what is required is a more balanced appreciation of the effect flexibility has on software. Yes it makes it adaptable in an uncertain climate but it also makes it more expensive to produce and harder to understand. As designers we need to begin to make more complicated value judgements about where on that spectrum we want our designs to be placed. We need to begin to answer questions such as, how much will this structure cost, versus the amount of future proofing it might provide. Only once we begin to make such decisions will software finally become a designable activity.

Conclusions

This paper has presented an overview of some of the design advice available for the object oriented paradigm. It has compared the work of Fowler and Beck, Riel and Meyer and found that each presents a diverse set of opinions of how software should be designed. It has presented details of each authors design advice and shown areas of similarity and areas of conflict in design. It has also contextualised this work by identifying the range of forces that should affect design decisions. It has shown that much of the current advice is restricted to supporting the flexibility, reusability and comprehension of software. While there is a need to continue to investigate these areas there is also a more pressing need to provide similar advice for factors such as cost, reliability and time to market.

The diversity on offer suggests that there is a continuing need to search for new design guidelines. The lack of overlap between authors suggests that they present windows into a much larger set of design advice. It is quite possible that the best is yet to come in terms of design guidelines. On the other hand there will also be an eventual need to validate the utility of the various design advice. This is a daunting prospect because there are a large number of heuristics and they all offer significant leeway in terms of how strictly they are implemented in a solution. This suggests that there will be a great many factors which might affect the final outcome. A study to investigate these claims will need to be sufficiently thorough and detailed to enable a fair evaluation to take place.

The ultimate goal of this work is more humble. It seeks to raise awareness of design heuristics by identifying potential areas of existing programs which violate design advice. This will hopefully enable developers to learn best practice retroactively and will also enable a practical form of evaluation where guidelines that are found to work well will likely come to be adopted by a designer in forward engineering tasks; a kind of survival of the fittest for design advice.

References

Beck, Kent. 2000. Extreme Programming Explained: Embrace change. Boston MA: Addison Wesley.

Fowler, Martin. 2000. Refactoring: Improving the design of existing code. Reading MA: Addison Wesley.

Ghezzi, Carlo, Mehdi Jazayeri and Dino Mandrioli. 2002. Fundamentals of Software Engineering. 2nd ed. Upper saddle river NJ: Prentice Hall.

Johnson, Ralph E. and Brian Foote. 1988. Designing Reusable Classes. Journal of Object Oriented Programming. 1(2): 22-35.

Lieberherr, Karl J. Ian M Holland. 1989. Assuring good style for object oriented programs. IEEE Software, September, 38-48.

Liskov, Barbara. 1988. Data abstraction and hierarchy. SIGPLAN notices. 23(5): 17-34.

Meyer, Bertrand. 1997. Object Oriented Software Construction. 2nd ed. Upper saddle river NJ: Prentice Hall.

Miller George A. 1957.The magic number seven, plus or minus two. The Psychological Review. 63: 81-97.

Ott, Linda and James M Bieman. 1998. Program Slices as an Abstraction for Cohesion Measurement. Information Science and Technology. 40(11-12): 691-700.

Parnas, David L. 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM. 15(12): 1053-1058.

Pressman, Roger S. 1994. Software Engineering: A practitioner’s approach. 3rd ed. London: McGraw Hill

Riel, Arthur J. 1996. Object Oriented Design Heuristics. Reading MA: Addison Wesley.

Somerville, Ian. 2001. Software Engineering. 6th ed. Harlow: Addison Wesley/Pearson

Wrifs-Brock, Rebecca. Brian Wilkerson and Lauren Wiener. 1990. Designing Object Oriented Software. Englewood Cliffs NJ: Prentice Hall.