Comp462 Essay 1 – Software development processes for the object-oriented paradigm.

Daniel Ballinger

300041839

Introduction

This essay examines the relationship between object-oriented (OO) development techniques and desirable characteristics of an object-oriented system. The software development techniques being considered are: use cases, essential use cases, CRC (Class, Responsibility, and Collaboration) cards, responsibility driven design (RDD), metrics, and heuristics. Each technique will be examined to see how it leads to an OO design and then to see how it encourages an OO design with desirable characteristics.

The different techniques fall into three broad categories. Use cases and essential use cases are used for requirements capture. CRC cards and RDD are techniques for guiding the design process. Metrics and heuristics are evaluation mechanisms that could be used for formative evaluation during the development process or for a summative evaluation.

I'll use the following as the main desirable characteristics/attributes of an object-oriented system:

- abstraction
- encapsulation
- inheritance
- polymorphism
- modularity
- reusability

Other desirable characteristics of a design process include:

- Maintainability
- Adaptability
- Scalability
- Testability
- Traceability
- Reliability
- Usability

(Concrete) Use Cases

Use cases and OO design

Use cases are an abstraction over a set of scenarios that describe the functionality of the system. They can be used during the early stages of a project during the requirements elicitation phase and then for traceability throughout the system development process. They help capture functionality required by the users of the system and provide a reasonable way to reach an agreement with the client about what should be developed. Essentially they are modelling tasks that the users want to be able to accomplish with the system (the functional requirements).

Use cases are often used to guide the design of communicating objects to satisfy the functional requirements of the system. Bruegge and Dutoit ([17] pg139) suggest objects in the design be derived from the use cases (and scenarios). Ivar Jacobson has also advocated relying on use cases as a way to elicit classes [11]. They suggest techniques to map use cases across to objects in the design including:

- Identifying entity, boundary and control objects using techniques like natural language analysis
- Mapping use cases to objects
- Identifying associations among objects and object attributes

Desirable OO characteristics and use cases

Define the scope of the system

With use cases it possible to identify functional requirements that are closely related to the system but do not fall within the scope of the project. This helps prevent a system design that extends to cover far more functionality than required.

Natural Language Problems

The quality and style of writing (verbification of nouns and consistency of terms) used by the analysts in producing the use cases will affect the quality of the design that emerges. The natural language analysis may produce many more nouns than relevant classes (nouns corresponding to attributes or synonyms for other nouns) and the designers must be able to remove unnecessary domain terms from objects, attributes, and associations.

Use cases are not object-oriented

In Object Oriented Software Construction [11](pg. 739), Bertrand Meyer is of the opinion that: "Except with a very experienced design team (having built successful systems of several thousand classes each in a pure O-O language), do not rely on use cases as a tool for object oriented analysis and design." His reasons for this opinion are based around arguments that inexperienced designers can introduce undesirable characteristics to a design when using use cases, or worse still, emulate functional programming concepts under the guise of OO design. Some of the ways this can happen are pointed out below.

Use cases emphasise ordering/sequencing and assume a sequential flow of control

When using use cases as the basis for the design of a project, the designers may also acquire the assumed sequential flow of control (the system must to a, then b,…). The emphasis placed on ordering early in design can create a system model that is fragile and subject to change, reducing the overall adaptability - [11]. A better approach is to look for the more abstract operations an object can perform and what are the high-level constraints on them (refer to Essential Use Cases).

If the physical sequence of operations is truly a requirement, Don Mills suggests that they are better captured more abstractly using preconditions - [3]. He points out that “Early emphasis on ordering is among the worst mistakes an OO project can make, but is hard to avoid if use cases are relied on for analysis…”
Use cases favour a functional approach, based on processes (actions).

Use cases themselves are not object-orientated as Luther Hampton points out in A Critique of Use Cases [16], “They are descriptions of interactions between the system and the users, and they say nothing about objects.” Instead they are procedural descriptions of functional decomposition - [3]. They embody an approach to design that resembles top-down functional design - [11]. This approach of specifying what the system does as the starting point is at odds with the OO concept of data and behavioural abstraction (what it does it to). This could reduce the adaptability of the system by tying it into a specific view of the internal workings too early.

Use cases depict how the users envisage the system

Uses cases have a tendency to depict the existing or envisaged processes that the users have of the system, computerised or not. This is undesirable as without a higher level of abstraction the system could end up perpetuating antiquated modes of operation. Clearly the system is being redesigned to address certain deficiencies, and propagating them along defeats the purpose of the new system. Essential use cases move to address this problem.

Essential use cases

Essential use cases and OO design

Essential use cases exist at a level of abstraction above (concrete) use cases. They attempt to capture the essence of the system away from implementation considerations and technology, but still convey the same crucial information. Like concrete use cases, essential use cases are first used during the early stages of requirements analysis (either instead of or in conjunction with). They can be derived from concrete use cases or used to create concrete use cases, although the latter process can be time consuming.

Through a process called usage-centered design [2] they can be used in the development of user interfaces. One of the benefits produced here is helping to remove users preconceptions about what the interface will look like.

It is also possible to use the essential use cases to directly drive the object-oriented design at the same time the user interface is being designed - [12]. In this method, concepts from CRC cards and RDD are borrowed allowing the essential use cases to identify a set of responsibilities that the system must fulfil. As in RDD (below), these responsibilities indicate the behaviour of the overall system but make no mention of how the system will be implemented. The collective responsibilities of the objects in the system must reflect the same behaviour identified by the essential use cases.

Essential use cases and desirable OO characteristics

One of the main benefits of essential use cases comes from the higher level of abstraction they provide. Moving away from specifying how the process should be carried out allows for a minimal internal design that meets the needs of users. This less binding approach allows the OO designers greater flexibility in meeting the requirements of the system and more opportunities to create innovative designs.

Benefits come from using responsibilities to link between the essential use cases and the objects. Firstly, the responsibilities provide bi-directional traceability between the essential use cases and objects - [12]. Secondly, the responsibilities do not bind the objects they are associated with into a particular implementation, which gives benefits resembling those from encapsulation (the internals of the object are hidden). One of these benefits is that changes in the way the system will be implemented will not effect the essential use cases.

CRC (Class, Responsibility, and Collaboration) cards

CRC cards and OO design

CRC cards are a technique used in the creation of an object-oriented system design. A class name, a set of responsibilities, and a set of collaborators used to meet these responsibilities, model each object in the design on a physical card. Much like RDD, the design is guided by an overall set of responsibilities the system must satisfy. These responsibilities are associated with objects and identify problems to be solved. Through an iterative process the arrangement and delegation of responsibilities and collaborators is adjusted until a satisfactory structure emerges.

Kent Beck and Ward Cunningham suggest "driving a design toward completion with the aid of execution scenarios" - [4]. As scenarios are the basis for concrete and essential use cases, they can also be used as to identify responsibilities that will drive the design.

CRC cards and desirable OO characteristics

Physical nature promotes easier understanding

The physical nature of the cards often leads to easier understanding of a design. Beck and Cunningham speculate that "because the designs are so much more concrete, and the logical relationship between objects explicit, it is easier to understand, evaluate and modify a design." - [4]. They also point out that the "ability to quickly organise and spatially address index cards proves most valuable when a design is incomplete or poorly understood." Also, an abstract set of objects "can be collected and handled as a single pile of cards with the most abstract card on top where it can represent the rest". This encourages the designer to think only about the particular abstraction and allows the refinements to be treated like the abstraction.

Avoiding global knowledge of control

Through physical interaction with the CRC cards, designers can identify with a single object and view the rest of the design from it's perspective. This helps them give up the global knowledge of control stemming from being able to view the entire system that is possible with procedural programs. This in turn promotes relying on only the local knowledge of objects to accomplish the tasks set out by the responsibilities - hence giving greater encapsulation and modularity.

Avoiding premature complexity

In CRC design, the design progresses from knowns to unknowns rather than top-down or bottom-up - [4]. This benefits the design by avoiding premature complexity, as the system will only meet the current demands and not mythical future needs. By ensuring that the system only contains as much information as the designer has directly experienced modularity and reusability are improved.

Using responsibilities, to state what collaborators do for the current object, prevents an inappropriate reliance on the implementation of other objects. Unnecessary relations between objects are avoided because an object will only ever reference another object if it needs it for a specific purpose. Also, in recording the motivation for collaboration, by representing (potentially) many messages as English phrases, it becomes easier for a designer to review the rationale for the design decisions.

Responsibility driven design (RDD)

RDD and OO design

Like CRC cards, RDD is a technique used in the creation of an object-oriented system design. Also like CRC cards, responsibilities identified during requirements elicitation are assigned to objects in the design. While CRC cards use the responsibilities associated with the objects to represent the problems to be solved, RDD uses them to represent the knowledge an object maintains or actions it can perform.

RDD and desirable OO characteristics

Designing classes without mention of implementation structure

One of the main advantages of RDD is that the structure of the objects is not included in the design and is postponed until the implementation stage. Instead the designer focuses on the contractual responsibilities of each object. To help achieve this, Rebecca Wirfs-Brock and Brian Wilkerson suggest RDD objects be based on the client-server model that focuses on what service the server supplies for the client but not how. This freedom from the premature definition of the internal structure of objects makes major design changes easier with less unnecessary information to consider - [5] (pg74).

In contrast to RDD, and demonstrating the undesirable characteristics that it avoids, data-driven design prescribes the structure of objects at an early stage. This risks the structure of the object becoming part of its definition and the operations evolving to reflect that structure (this is particularly a problem with procedural design, which data-driven design is similar to - [5]. Attempts to change the structure of the object transparently are destined to fail because other classes rely on that structure. Hence encapsulation has been violated on both the structure and behaviour of the object.

Another benefit of using both the client-server model and RDD is that, not only does an object not need to know how the collaborating object implements a service being requested, it doesn't know what object type is responding to the request. This property can be used to identify standard protocols (message names) between objects, facilitating polymorphism - [5].

Abstract classes and the inheritance hierarchy

Constructing abstract classes can be difficult in a data-driven design as they must be found and extracted from a concrete set of classes - [5]. Maintaining contracts (the agreements between client and server) becomes difficult if the clients have become dependant on the implementation of the server, which Wirfs-Brock and Wilkerson point out they often do. RDD's greater encapsulation makes lifting out abstract classes easier, as parts of the interface of existing objects relating to implementation will not be present yet - removing the unnecessary dependencies on the server.

As only the types of classes are known during design in RDD, the inheritance hierarchy is encouraged to follow a type hierarchy. Wirfs-Brock and Wilkerson observe that this will "improve encapsulation with respect to subclass clients, ensuring that all inherited behaviour is part of the contract of the class."

Metrics

Metrics and OO design

Metrics are evaluation tools that can be used for measuring the quality of an OO design and for performing comparative assessments between designs. To do this, they assess designs in ways that aim to quantify desirable characteristics and gauge their complexity.

In order to produce an informative evaluation of an OO design, metrics must be targeted at desirable OO characteristics. Shyan Chidamber and Chris Kemerer suggest that this alone is not enough, and that the metrics used should have a solid theoretical base in measurement theory - [8]. This theory would aim to reflect the viewpoints of experienced OO software developers and evaluate the metrics themselves against software metric evaluation criteria. They propose that this will help move software development from its current 'craft' into something resembling conventional engineering.

Metrics and desirable OO characteristics

Metric	**Abbreviation**	**Desirable characteristics they can examine**
Weighted methods Per Class	(WMC)	reusability
Depth of Inheritance Tree	(DIT)	inheritance, reusability
Number of children	(NOC)	inheritance, reusability
Response for a Class	(RFC)	complexity
Lack of cohesion in methods	(LCOM)	encapsulation, complexity
Coupling between objects	(CBO)	modularity, reusability

Table 1 – Candidate OO metrics listed in [8] by Shyan Chidamber and Chris Kemerer

The metrics mentioned in table 1 aim to examine the definition, attributes and communication elements of objects in a design. Chidamber and Kemerer explain in [8] how these metrics can measure the degree to which a design exhibits the desirable characteristics listed in table 1. For example, they explain that that LCOM is undesirable as cohesion promotes encapsulation, and that a lack of cohesiveness implies an object should be split.

<u>Evaluate the degree of object orientation.</u>

Applying a suitable set of metrics to a particular system can reveal the degree of object-orientation that it exhibits. This in turn can be used to improve the quality of the design by identifying areas where it shows undesirable characteristics. Identifying these areas may have to be done comparatively to similar scale projects or rely on the experience of those applying the metrics to identify them as undesirable results.

<u>Learning tool for programmers new to OO</u>

For developers new to OO, with or without other programming background, there is a danger that they may program using a procedural/functional style in an OO language (like C++ or Java). Metrics can be used to help train current and new software engineers in generally accepted OO principles by identifying and hence avoiding poor design - [8].

Heuristics

Heuristics and OO design

Heuristics aim to capture principles that good OO designers would use in assessing a design. Less formally, they represent the 'it feels right' concepts that a guru would run through subconsciously in analysing a design - [9]. In capturing these desirable features they can be communicated to other designers who can then apply them to their work and get the benefits of the experienced designers understanding.

Like metrics, heuristics can be used as an evaluation tool and Jacob Nielsen demonstrates this in [10] doing usability testing. However, as heuristics are not measurement based, they can also be applied less formally during many stages of design. Ideally the designers of the system would be familiar with most of the heuristics, if not all, before starting the design so they could influence them at some level during development. The advantages of this are that potential problems are more likely identified and addressed earlier rather than later. For those problems that do elude the designers, a heuristic evaluation can help detect them.

Heuristics and desirable OO characteristics

Rather than leading to desirable OO characteristics, heuristics are desirable OO characteristics, both in terms of characteristics to strive to enhance and those to avoid. The advantages brought to OO design are as vast as the issues they address. Arthur Riel, in Object-Oriented Design Heuristics - [9], identifies approximately 60 heuristics addressing issues including abstraction, encapsulation, modularity, reusability, minimal coupling, avoiding god classes, modelling a problem with too many classes, and modelling things outside the system.

Different heuristics have different priorities for different designs. These priorities will be defined by a combination of the application domain and the users needs. Riel points out that "The strength of a heuristic comes from the ramification of violating it." Riel's description of each design heuristic includes a justification that benefits both new and experienced designers - [9]. While explaining important OO concepts to new designers, these justifications also document an informal list of "Consequences" for experienced designers. When making tradeoffs on conflicting physical and logical design requirements, the consequences of violating certain heuristics can help justify decisions.

Some of Riel's heuristics can also be used to tell a designer when it is time to apply one of several design patterns (a reusable part of design to that has been proven to solve a particular problem) - [9]. This helps address one of the problems with patterns, which is identifying when they should be used.

Conclusions

A design technique should promote good design, teach new designers how to design well, and help standardise the way designs are developed. When using design techniques to produce an object-oriented model, the results should exhibit desirable object-oriented characteristics.

The six techniques above indicated that they all had the ability to contribute to a good object-oriented design. However, it is important that the designers understand the issues involved in using each technique. In many cases, experience was the best way to ensure this.

Use cases are an effective tool in capturing the requirements of the system. But there non-object-oriented nature can introduce undesirable characteristics if the designer does not acknowledge their emphasis on ordering and functional decomposition of problems.

Essential use cases can address many of the problems the concrete use cases exhibit by removing the emphasis form ordering and technology, and instead looking at the essence of the task.

CRC cards and RDD use responsibilities in slightly differing ways to avoid premature complexity in the design. Both defer implementation details to later stages to facilitate stronger encapsulation of objects and opportunities to look for better abstractions.

Metrics and heuristics provide a way to communicate the understanding of experienced designers to other designers. Both have applications in evaluation and aim to record what it means for a design to be good. Heuristics also have the ability to encourage justification in design decisions.

An overall theme that can be seen in the techniques is that it is important not to incorporate the next design phase prematurely in the current design phase. The technique should be focused on only performing the current phase. For example, the requirements elicitation phase should not put the horse before the cart and presume the structure of the program unless it really is a requirement.

References

[1] Martin Fowler: Chapter 3 from UML Distilled: *Applying the Standard Object Modeling Language*, Addison-Wesley 1997,2000. http://www.awl.com/cseng/titles/0-201-32563-2/umldist-chap3.html

[2] Larry Constantine and Lucy Lockwood: *Structure and Style in Use Cases for User Interface Design from Object-Modeling and User Interface Design* van Harmelen (ed.), Addison-Wesley, 2001, http://www.foruse.com, http://www.foruse.com/Files/Papers/structurestyle2.pdf

[3] Don Mills: *What's the Use of a Use Case* ACM OOPSLA 2001 Workshop on Behavioural Semantics

[4] Kent Beck and Ward Cunningham: *A Laboratory for Teaching Object-Oriented Thinking*, Proceedings of OOPSLA 1989. http://c2.com/doc/oopsla89/paper.html

[5] Wirfs-Brock, Rebecca and Wilkerson, Brian: *Object-Oriented Design: A responsibility-driven approach*, Proceedings of OOPSLA 1989.

[6] Rebecca Wirfs-Brock, Brain Wilkerson, and Lauren Wiener: *Designing Object-Oriented Software*, Prentice-Hall, 1990.

[7] Rebecca Wirfs-Brock, How Designs Differ, *Report on Object Analysis and Design*, Volume 1, Number 4.

[8] Shyam R. Chidamber, Chris F. Kemerer: *Towards a Metrics Suite for Object Oriented Design*, Proceedings of OOPSLA 1991.

[9] Arthur Riel: Extract (Preface) from *Object-Oriented Design Heuristics*, Addison Wesley 1996.

[9b] Arthur Riel: Extract (Chapter 3) from *Object-Oriented Design Heuristics*, Addison Wesley 1996.

[10] Jacob Nielsen: *How to Conduct a Heuristic Evaluation*, http://www.useit.com

[11] Bertrand Meyer: *OOSC2: The Use Case Principle*, *Extract from Object Oriented Software Construction 2nd Edition*, http://www.elj.com/elj/v1/n2/bm/use-cases/

[12] James Noble, Robert Biddle, and Ewen Tempero: *Essential Use Cases and Responsibility in Object-Oriented Development*, Victoria University of Wellington, 2001. http://www.foruse.com/Files/Papers/euc-responsibility.pdf, http://www.mcs.vuw.ac.nz/comp/Research/object/Papers/euc-html

[13] Russell C. Bjork: *An Example of Object-Oriented Design: An ATM Simulation*, http://inprem.rug.ac.be/~gpremer/OOA/ATM_Example/default.html

[14] Ralph E. Johnson, Brian Foote: *Designing Reuseable Classes*, University of Illinois.

[15] Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Övergaard: *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Wokingham (England), 1992.

[16] Luther Hampton, Robert C. Martin, Frank G. Pitt, Tim Ottinger: *A Critique of Use Cases*, 9 July 1997. http://ootips.org/use-cases-critique.html

[17] Bernd Bruegge and Allen H Dutiot: *Object-Oriented Software Engineering*, Prentice Hall (2000).

[18] Chandra Vemulapalli: *A Use Case FAQ(Frequently Asked Questions)*, Advanced Software Technologies Group, WorldCom Inc., http://www.unantes.univnantes.fr/usecase/Contributions/chandra.html

[19] Warren Young: *Improving Object-Oriented Designs,* Dr. Dobb's Journal, October, 1996 http://www.ercb.com/ddj/1996/ddj.9610.html

[foldoc] Denis Howe: *Free on-line dictionary of computing,* Imperial College Department of Computing (1993). http://www.foldoc.org

Glossary

These terms are defined using quotes from references. There is some variance in the meaning of each term between different references and some of the quotes are benefits of using the concept the term represents.

Abstraction:
[foldoc] “Generalisation; ignoring or hiding details to capture some kind of commonality between different instances.”
[14] “Data abstraction encourages modular systems that are easy to understand.”

Encapsulation:
[foldoc] “The ability to provide users with a well-defined interface to a set of functions in a way which hides their internal workings. In object-oriented programming, the technique of keeping together data structures and the methods (procedures) which act on them.“
[5] “The ability to provide users with a well-defined interface to a set of functions in a way which hides their internal workings. “
[14] “Polymorphism makes it easier for a given component to work correctly in a wide range of new contexts.”

Inheritance:
[foldoc] “Methods or code in one class can be passed down the hierarchy to a subclass or inherited from a superclass.”

Polymorphism:
[foldoc] “In object-oriented programming, the term is used to describe variables which may refer at run time to objects of different classes.”

Modularity:
[www.dictionary.com] “Designed with standardised units or dimensions, as for easy assembly and repair or flexible arrangement and use: *modular furniture; modular homes.*“
The program/design is divided into components, each component is complete in itself and with its own function.

Reusability:
[foldoc] “Using code developed for one application program in another application. Traditionally achieved using program libraries. Object-oriented programming offers reusability of code via its techniques of inheritance and genericity.”

Scenario:
[2] “Scenarios are typically extended narratives forming a plausible vignette or story-line. They tend to be rich, realistic, concrete, and specific, often replete with gratuitous detail for enhanced verisimilitude.”
[17] “Instance of a use case. A scenario represents a concrete sequence of interactions between one or more actors and the system.”

Use cases:
[1] - “In essence, a use case is a [single] typical interaction between a user and a computer system.” “Use cases are all about externally-required functionality.”
[11] - “a *complete course of events initiated by a* [user of the future system] *and* [of] *the interaction between* [the user] *and the system.* “
[2] (Jacobson’s original definition) - “A use case is a specific way of using the system by using some part of functionality. [A use case] constitutes a complete course of interaction that takes place between an actor and the system.”
[2] (Developers of UML) - “The specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside actors.
[2] (M Fowler) - “A use case is a typical interaction between a user and a computer system … [that] captures some user-visible function … [and] achieves a discrete goal for the user “.
[18] (From Jacobson) - “a good use case as one, which performs some action in the system, that is of a *measurable value* for a *particular actor*”.

Essential use cases:
[12] “Essential use cases are abstract, lightweight, technology-free dialogues of user intention and system responsibility, that effectively capture requirements for user interface design.”
[2] “… abstract, generalised, and technology-free descriptions of the essence of a problem.”
[2] “… a single, discrete, complete, meaningful, and well-defined task of interest to an external user in some specific role or roles in relationship to a system, comprising the user intentions and system

responsibilities in the course of accomplishing that task, described in abstract, technology free, implementation-independent terms using the language of the application domain and of external users in role."
[2] "… they model tasks in a form closest to the essential nature of a problem and do not intermingle design solutions with problem description."

Use case **goal** (user goal):
[2] "A goal is the desired end state of the system, and as such it is correctly described in static terms as the state and features of objects." "the destination"
[2] "Goals, being static, place the focus on objects or nouns,…"

Use case **intention** (system interaction):
[2] "An intention is dynamic and represents direction or progress rather than an end state." "represent the journey – the interaction"
[2] "…intentions, being dynamic, bring the actions and verbs to the foreground, …"

Entity objects:
"Represent the persistent information in the system."

Control objects:
"Represent the user tasks the system supports."

Boundary objects:
"represent system/actor interactions."

Actor:
[17] "External entity that needs to exchange information with the system. An actor can represent either a user role or another system."
[1] "An actor is a role that a user plays with respect to the system."

CRC (Class, responsibility, and collaboration) card:
A card which records the class name, Responsibilities to be fulfilled by the class, and Collaborators (Helper Objects to the class).
[4] "… characterise objects by class name, responsibilities, and collaborators, …"
[6] "… used to capture information about the classes and subsystems in a design."
[4] "CRC cards place the designer's focus on the motivation for collaboration by representing (potentially) many messages as a phrase in English text"

Responsibility-Driven Design (RDD):
[5] "Responsibility-driven design is inspired by the client/server model. It focuses on the contract by asking: * What actions is this object responsible for? And * What information does this object share?" "Information shared may or may not be part of the object. First focus on fully specifying the desired behaviour and associated sets of responsibilities for each object. We focus on structure during implementation."

Responsibility:
[4] "Responsibilities identify problems to be solved." "Serves as a handle for discussing potential solutions."

Collaborators:
[4] objects that "will send or be sent messages in the course of satisfying responsibilities."

Heuristic:
[www.dictionary.com] "Relating to or using a problem-solving technique in which the most appropriate solution of several found by alternative methods is selected at successive stages of a program for use in the next step of the program."
[9] "guidelines to help developers make proper decisions." "rules of thumb."
[19] "A heuristic is similar to a rule ("A class should capture one, and only one, key abstraction."), but isn't intended to be inviolate. Instead, it acts as a guideline that generally indicates the correct way to design something."

Metrics:
[foldoc] "A measure of software quality which indicates the complexity, understandability, testability, description and intricacy of code."

Coupling:
[8] "The degree to which an object acts upon the other."
Minimal amount of coupling between objects is desirable. Promotes encapsulation.

Cohesion:
[8] “The degree of similarity of methods relates both to the conventional notion of cohesion in software engineering, (i.e., keeping related things together) as well as encapsulation of objects, that is, the bundling of methods and instance variables in an object. Cohesion of methods can be defined to be the *degree of similarity* of methods. The higher the degree of similarity of methods, the greater the cohesiveness of the methods and the higher the degree of encapsulation of the object.”
Strong cohesion within an object is desirable. Promotes encapsulation. Lower cohesion increases complexity.

Object-oriented design (OOD):
[8] “Booch (1986) defines object oriented design to be the process of identifying objects and their attributes, identifying operations suffered by and required of each object and establishing interfaces between objects.”
[9] “object-oriented development focuses on a decentralised collection of co-operating entities.”
[9b] “It is this decentralisation of software that gives the object-oriented paradigm its ability to control essential complexity.”

Class:
[9b] “the bi-directional relationship between data and behaviour, namely, a class, in object-oriented terminology.”
[foldoc] “A set of objects which share a common structure and behaviour.”