# REFERENCES


(1) ISO, International Organization for Standardization. ISO 9126-1:2001, Software engineering – Product quality, Part 1: Quality model. 2001.

(2) Page-Jones, M. The Practical Guide to Structured System Design. Yourdon Press, 1980.

(3) Boehm, B., Brown, J., Kaspar, H., Lipow, M., McLeod, G., and Merritt, M. Characteristics of Software Quality, North Holland, 1978.

(4) Marinescu, R. and Ratiu, D. Quantifying the Quality of Object-Oriented Design: The Factor-Strategy Model. Proceeding of 11th Working Conference on Reverse Engineering (WCRE), 2004: 192 – 201.

(5) McCall, J., Richards, P., and Walters, G. Factors in Software Quality, Volumes I, II, and III, US. Rome Air Development Center Reports NTIS AD/A-049 014, NTIS AD/A-049 015 and NTIS AD/A-049 016, U. S. Department of Commerce, 1977.

(6) Gregor, K., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin J. Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming, 1997: 220–242.

(7) Jindasawat, N., Kiewkanya, M. and Muenchaisri, P. Investigating Correlation between the Object-Oriented Design Maintainability and Two Sub-Characteristics. Proceeding of 13th International Conference on Intelligent & Adaptive Systems, and Software Engineering, 2004: 151-156.

(8) Genero, M., Paittini, M. and Manso, E. Finding 'Early' Indicators of UML Class Diagrams Understandability and Modifiability. Proceedings of the 2004 International Symposium on Empirical Software Engineering, 2004.

(9) Sheldon, F., Jerath, K., and Chung, H. Metrics for maintainability of class inheritance hierarchies, Journal of Software Maintenance and Evolution: Research and Practice. 14(2002): 147-160.

(10) Sant'Anna, S., Garcia, A., Chavez, C., Lucena, C., and Arndt von Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment

Framework. Proceedings of Brazilian Symposium on Software Engineering (SBES'03), 2003, 19-34.

(11) Bartsch, M. and Harrison, R. An exploratory study of the effect of aspect-oriented programming on maintainability. Software Quality Control. 16, 1(2008): 23 – 44.

(12) Laddad, R. AspectJ in Action: Practical Aspect-Oriented Programming. Manning, 2003.

(13) Zhao, J., Dependence Analysis of Aspect-Oriented Software and Its Applications to Slicing, Testing, and Debugging, Technical-Report SE-2001-134-17, Information Processing Society of Japan (IPSJ), 2001.

(14) Deligiannis, I., Shepperd, M., Stamelos, I. and Roumeriotis, M. An empirical investigation of an object-oriented design heuristic for maintainability, Journal of Systems & Software. 65, 2(2003): 127-139.

(15) Fowler, M. Refactoring: Improving the Design of Existing Code, Addison-Wesley, 2000.

(16) Monteiro, M. and Fernandes, J. Towards a Catalogue of Refactorings and Code Smells for AspectJ, Transactions on Aspect-Oriented Software Development. (2006): 214 – 258.

(17) Piveta, E., Hecht, M., Pimenta, M. and Price, R. Detecting Bad Smells in AspectJ, Journal of Universal Computer Science, 12, 7(2006): 811-827.

(18) Srivisut, K. and Muenchaisri, P. Defining and Detecting Bad-Smells of Aspect-Oriented Software. Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC), 2007.

(19) Zhang, S. and Zhao, J. Defining and Detecting Bad-Smells of Aspect-Oriented Software. Proceedings of the 31st Annual International Computer Software and Applications Conference, 2007.

(20) Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

(21) Chidamber, S.R. and Kemerer, C.F. A metric suit for object-oriented design. IEEE Trans. on Software Engineering. 20,6(1994): 476-493.

(22) Li, W. Another metric suite for object-oriented programming. The Journal of Systems and Software. 44, 2(1998): 155-162.

(23) Sheldon, F. T., Jerath, K. and Chung, H. Metrics for Maintainability of Class Inheritance Hierarchies. Journal of Software Maintenance and Evolution: Research and Practice. 14,1(2002):147-160.

(24) Wampler, D. Aspect-Oriented Design Principles: Lessons from Object-Oriented Design, The industry track of International Conference on Aspect-Oriented Software Development, 2007.

(25) Zhao, J. Measuring Coupling in Aspect-Oriented Systems, International Software Metrics Symposium (METRICS), 2004.

(26) Zhao, J. and B. Xu. Measuring Aspect Cohesion. Proceeding of International Conference on Fundamental Approaches to Software Engineering (FASE), 2004.

(27) Zhao, J., Towards a Metrics Suite for Aspect-Oriented Software, Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), 2002.

(28) Kang, D., Xu, B., Lu, J. and Chu, W. C. A Complexity Measure for Ontology Based on UML. Proceedings of the 10th IEEE International Workshop on Future Trends of Distrubuted Computing Systems, Suzhou, Chaina, May 2004:222-228.

(29) Filman, R. and Friedman, D. Aspect-Oriented Programming is Quantification and Obliviousness. Aspect-Oriented Software Development, Addison-Wesley, 2005: 21-31.

(30) Noda N. and Kishi T. On Aspect-Oriented Design - Applying Multi-Dimensional Separation of Concerns on Designing Quality Attributes. Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems, 1999.

(31) Eaddy, M., Aho, A., and Murphy, G. Identifying, Assigning, and Quantifying Crosscutting Concerns. ICSE Workshop on Assessment of Contemporary Modularization Techniques (ACoM), Minneapolis, MN, USA, 2007.

(32) Hannemann, J. and Kiczales, G. Design pattern implementation in Java and AspectJ. Proceedings of the 17th Annual ACM conference on Object-

Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002.

(33) Murphy, G., Walker, R., Baniassad, E., Robillard, M., Lai, A. and Kersten, M. Does Aspect-oriented Programming Work? Communications of the ACM, Special Issue on Aspect-Oriented Programming, 44, 10(2001): 75-77.

(34) Soares, S., Laureano, E., and Borba, P. Implementing distribution and persistence aspects with AspectJ. Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002.

(35) Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., and Staa, A. Modularizing Design Patterns with Aspects: A Quantitative Study. LNCS Transactions on Aspect-oriented Software Development (TAOSD), 2005: 36-74.

(36) Riel, A. Object-Oriented Design Heuristics, Addison-Wesley, 1996.

(37) Metsker, S., and Wake, W. Design Patterns in Java, Addison-Wisely, 2006.

(38) Allchin, D. Error Types. Perspectives on Science. 9 (2001): 38 – 58.

(39) Trifu, A. and Marinescu, R. Diagnosing Design Problems in Object Oriented Systems, Proceeding of 12th IEEE Working Conference on Reverse Engineering (WCRE), IEEE Computer Society Press, 2005

(40) Piveta, E., Hecht, M., Pimenta, M., and Price, R. T. Bad smells em sistemas orientados a aspectos (in portuguese). Proceedings of Brazilian Symposium on Software Engineering (SBES 2005), 2005.

(41) Hachani, O. and Bardou, D. Using Aspect-Oriented Programming for Design Patterns Implementation. Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS), 2002.

(42) Eclipse. The AspectJ Programming Guide, 2001, http://www.eclipse.org/aspectj/doc/released/progguide/index.html/

(43) Evertsson, G. Tetris in AspectJ, 2003, http://www.guzzzt.com/coding/aspecttetris.shtml

(44) Chernoff, H. and Lehmann, E. The use of maximum likelihood estimates in $\chi^2$ tests for goodness-of-fit, The Annals of Mathematical Statistics, 25(1954): 579 – 586.

(45) Ceccato, M. and Tonella, P. Measuring the Effects of Software Aspectization. Proceedings of the first Workshop on Aspect Reverse Engineering, 2004.

(46) Stochmialek, M. AOP Metrics, 2006, http://aopmetrics.tigris.org/metrics.html

(47) Kiewkanya, M. Measuring Object-Oriented Software Maintainability in Design Phase Using Structural Complexity and Aesthetic Metrics, PhD thesis, Chulalongkorn University, Thailand, 2006.

(48) Bradley, J. An Examination of Aspect-Oriented Programming in Industry, Technical Report CS-03-108, Department of Computer Science, Colorado State University, Fort Collins, Colorado, 2003.

(49) Kremer, R. Practical Software Engineering, 1998, http://sern.ucalgary.ca/courses/cpsc/451/W98/Complexity.html#RTFToC120/

(50) Rumbaugh, J., Booch, G. and Jacobson, I. Unified Modeling Language Reference Manual, Addison-Wesley, 1998.

# APPENDICES

# APPENDIX A

# DESIGN PRINCIPLES SUMMARY

## Aspect-Oriented Design Guidelines Summary

### Design Guidelines for Pointcuts and Advices

*Guideline 1:* Don't make the core concern unused by replacing the entire of its context with the crosscutting concern.
*Guideline 2:* Do suitably use wildcard operators for each captured join point to increase the pointcut readability and to decrease the unintended join point capture.
*Guideline 3:* Do define a concise pointcut for capturing sibling join points to decrease the accidental join point miss when a new class is added to the hierarchy.

### Design Guidelines for Aspects

*Guideline 4:* Don't use an aspect privilege to leak the secret of other types.
*Guideline 5:* Do create an advantageous aspect which crosscuts at least two points in the core concern.

### Design Guidelines for the Relationships Between Aspects and Other Components

*Guideline 6:* Do localize all crosscutting contexts within a set of corresponding aspects.
*Guideline 7:* Do try to separate one set of aspects to be independent of another.
*Guideline 8:* Do minimize a number of aspects which crosscut to the same concern.
*Guideline 9:* Do keep scattering behaviors that represent the same concern in one set of corresponding aspects.
*Guideline 10:* Don't let a base class be dependence on its crosscutting aspects.

### Design Guidelines for Inheritance Relationships

*Guideline 11:* Don't use an inheritance relationship to model a crosscutting relationship because an aspect is not a special type of the class that it crosscuts.
*Guideline 12:* Don't introduce methods to all subclasses to override a concrete method of their abstract superclass.
*Guideline 13:* Don't define methods of all subaspects to override a concrete method of their abstract aspect.
*Guideline 14:* Do extract an abstract aspect if the same crosscutting concern has two or more variations.
*Guideline 15:* Don't introduce a method which has the same signature as an inherited method to the realized interface.

## Object-Oriented Design Heuristics Summary

**Heuristics for class and objects.**

*Heuristic 2.1:* All data should be hidden within its class.
*Heuristic 2.2:* Users of a class must be dependent on its public interface, but a class should not be dependent on its users.
*Heuristic 2.3:* Minimize the number of messages in the protocol of a class.
*Heuristic 2.4:* Implement a minimal public interface that all classes understand [e.g., operations such as copy, equality testing, pretty printing, etc.].
*Heuristic 2.5:* Do not put implementation details such as common-code private functions into the public interface of a class.
*Heuristic 2.6:* Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.
*Heuristic 2.7:* Classes should only exhibit nil or export coupling with other classes, that is, a class should only use operations in the public interface of another class or having not to do with that class.
*Heuristic 2.8:* A class should capture one and only one key abstraction.
*Heuristic 2.9:* Keep related data and behavior in one place.
*Heuristic 2.10:* Spin off non-related information into another class.
*Heuristic 2.11:* Be sure the abstractions that you model are classes and not simply the roles objects play.

**Heuristics for topologies of action-oriented versus object-oriented applications.**

*Heuristic 3.1:* Distribute system intelligence horizontally as uniformly as possible, that is, the top-level classes in a design should share the work uniformly.
*Heuristic 3.2:* Do not create god classes/objects in your system. Be very suspicious of a class whose name contains *Driver*, *Manager*, *System*, or *Subsystem*.
*Heuristic 3.3:* Beware of classes that have many accessor methods defined in their public interface. Having many implies that related data and behavior are not being kept in one place.
*Heuristic 3.4:* Beware of class that have too much non-communicating behavior, that is, methods that operate on a proper subset of the data members of a class. God class often exhibit a great deal of non-communicating behavior.
*Heuristic 3.5:* In applications that consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.
*Heuristic 3.6:* Model the real world whenever possible (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behavior in one place)
*Heuristic 3.7:* Eliminate irrelevant classes from your design.
*Heuristic 3.8:* Eliminate classes that are outside the system.
*Heuristic 3.9:* Do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb, especially those which have only one piece of meaningful behavior. Ask if that piece of meaningful behavior needs to be migrated to some existing or undiscovered class.
*Heuristic 3.10:* Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

**Heuristics for the relationships between classes and objects.**

*Heuristic 4.1:* Minimize the number of classes with which another class collaborates.
*Heuristic 4.2:* Minimize the number of message sends between a class and its collaborator.
*Heuristic 4.3:* Minimize the amount of collaboration between a class and its collaborator, that is, the number of different messages sent.
*Heuristic 4.4:* Minimize fan-out in a class, that is, the product of the number of message defined by the class and the message they send.
*Heuristic 4.5:* If a class contains objects of another class, then the containing class should be sending messages to the contained objects, that is, the containment relationship should always imply a uses relationship.
*Heuristic 4.6:* Most of the methods defined on a class should be using most of the data members most of the time.
*Heuristic 4.7:* Classes should not contain more objects than a developer can fit in his or her short-term memory. A favorite value of this number is six.
*Heuristic 4.8:* Distribute system intelligence vertically down narrow and deep containment hierarchies.
*Heuristic 4.9:* When implementing semantic constraints, it is best to implement them in terms of the class definition. Often this will lead to a proliferation of classes, in which case the constraint must be implemented in the behavior of the class-usually, but not necessarily, in the constructor.
*Heuristic 4.10:* When implementing semantic constraints in the constructor of a class, place the constraint test in the constructor as far down a containment hierarchy as domain allows.
*Heuristic 4.11:* The semantic information on which a constraint is based is best placed in a central, third-party object when that information is volatile.
*Heuristic 4.12:* The semantic information on which a constraint is based is best decentralized among the classes involved in the constraint when that information is stable.
*Heuristic 4.13:* A class must know what it contains, but it should never know who contains it.
*Heuristic 4.14:* Objects that share lexical scope-those contained in the same containing class should not have uses relationships between them.

**Heuristics for the inheritance relationship.**

*Heuristic 5.1:* Inheritance should be used only to model a specialization hierarchy.
*Heuristic 5.2:* Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.
*Heuristic 5.3:* All data in a base class should be private; do not use protected data.
*Heuristic 5.4:* In theory, inheritance hierarchies should be deep-the deeper, the better.
*Heuristic 5.5:* In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six.
*Heuristic 5.6:* All abstract classes must be base classes.
*Heuristic 5.7:* All base classes should be abstract classes.
*Heuristic 5.8:* Factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy.
*Heuristic 5.9:* If two or more classes share only common data (no common behavior), then that common data should be placed in a class that will be contained by each sharing class.
*Heuristic 5.10:* If two or more classes have common data and behavior, then those

classes should each inherit from a common base class that captures those data and methods.

*Heuristic 5.11:* If two or more classes share only a common interface, then they should inherit from a common base class only if they will be used polymorphically.

*Heuristic 5.12:* Explicit case analysis on the type of an object is usually an error. The designer should use polymorphism in most of these cases.

*Heuristic 5.13:* Explicit case analysis on the value of an attribute is often an error. The class should be decomposed into an inheritance hierarchy, where each value of the attribute is transformed into a derived class.

*Heuristic 5.14:* Do not model the dynamic semantics of a class through the use of the inheritance relationship. An attempt to model dynamic semantics with a static semantic relationship will lead to a toggling of types at runtime.

*Heuristic 5.15:* Do not turn objects of a class into derived classes of the class. Be very suspicious of any derived class for which there is only one instance.

*Heuristic 5.16:* If you think you need to create new classes at runtime, take a step back and realize that what you are trying to create are objects. Now generalize these objects into a class.

*Heuristic 5.17:* It should be illegal for a derived class to override a base class method with a NOP method, that is, a method that does nothing.

*Heuristic 5.18:* Do not confuse optional containment with the need for inheritance. Modeling optional containment with inheritance will lead to a proliferation of classes.

*Heuristic 5.19:* When building an inheritance hierarchy, try to construct reusable frameworks rather than reusable components.

**Heuristics for the multiple inheritance.**

*Heuristic 6.1:* If you have an example of multiple inheritance in your design, assume you have made a mistake and then prove otherwise.

*Heuristic 6.2:* Whenever there is inheritance in an object-oriented design, ask your self two questions:

1. Am I a special type of the thing from which I'm inheriting?
2. Is the thing from which I'm inheriting part of me?

*Heuristic 6.3:* Whenever you have found a multiple inheritance relationship in an object-oriented design, be sure that no base class is actually a derived class of another base class.

**Heuristic for the association relationship.**

*Heuristic 7.1:* When given a choice in an object-oriented design between a containment relationship and association relationship, choose the containment relationship.

**Heuristic for class-specific data and behavior.**

*Heuristic 8.1:* Do not use global data or functions to perform bookkeeping information on the objects of a class. Class variables or methods should be used instead.

**Heuristics for physical object-oriented design.**

*Heuristic 9.1:* Object-oriented designers should not allow physical design criteria to corrupt their logical designs. However, physical design criteria often are used in the decision-making process at logical design time.

*Heuristic 9.2:* Do not change the state of an object without going through its public interface.

## Object-Oriented Bad Smells Summary

*Duplicated Code:* The same code structure in two or more places is a good sign that the code needs to refactored: if you need to make a change in one place, you'll probably need to change the other one as well, but you might miss it.

*Long Method:* Long methods should be decomposed for clarity and ease of maintenance.

*Large Class:* Classes that are trying to do too much often have large number of instance variables. Sometimes groups of variables can be clumped together. Sometimes they are only used occasionally. Over-large classes can also suffer from code duplication.

*Long Parameter List:* Long parameter lists are hard to understand. You don't need to pass in everything a method needs, just enough so it can find all it needs.

*Divergent Change:* Software should be structured for ease of change. If one class is changed in different ways for different reasons, it may be worth splitting the class in two so each one relates to a particular kind of change.

*Shotgun Surgery:* If a type of program change requires lots of little code changes in various different classes, it may be hard to find all the right places that do need changing.

*Feature Envy:* This is where a method on one class seems more interested in the attributes (usually data) of another class than in its own class.

*Data Clumps:* Sometimes you see the same bunch of data items together in various places: fields in a couple classes, parameters to methods, local data. Maybe they should be grouped together into a little class.

*Primitive Obsession:* Sometimes it's worth turning a primitive data type into a lightweight class to make it clear what it is for and what sort of operations are allowed on it (e.g. creating a date class rather than using a couple of integers).

*Switch Statements:* Switch statements tend to cause duplication. You often find similar switch statements scattered through the program in several places. If a new data value is added to the range, you have to check all the various switch statements. Maybe classes and polymorphism would be more appropriate.

*Parallel* Inheritance *Hierarchies:* In this case, whenever you make a subclass of one class, you have to make a subclass of another one to match.

*Lazy Class:* Classes that are not doing much useful work should be eliminated.

*Speculative Generality:* Often methods or classes are designed to do things that in fact are not required. The dead-wood should probably be removed.

*Temporary Field:* It can be confusing when some of the member variables in a class are only used occasionally.

*Message Chains:* A client asks one object for another object, which is then asked for another object, which is then asked for another, etc. This ties the code to a particular class structure.

*Middle Man:* Delegation is often useful, but sometimes it can go too far. If a class is acting as a delegate, but is performing no useful extra work, it may be possible to remove it from the hierarchy.

*Inappropriate Intimacy:* This is where classes seem to spend too much time delving into each other's private parts. Time to throw a bucket of cold water over them!

*Alternative classes with different interfaces:* Classes that do similar things, but have different names, should be modified to share a common protocol.

*Incomplete Library Class:* It's bad form to modify the code in a library, but sometimes they don't do all they should do.

*Data Class:* Classes that just have data fields, and access methods, but no real behavior. If the data is public, make it private!

*Refused Bequest:* If a subclass doesn't want or need all of the behavior of its base class, maybe the class hierarchy is wrong.

*Comments:* If the comments are present in the code because the code is bad, improve the code.

## Aspect-Oriented Bad Smells Summary

*Anonymous Pointcut Definitions:* The use of the pointcut definition predicate directly in an advice may reduce legibility and hide the predicate's intention.

*Lazy Aspects:* if an aspect has few responsibilities, and its elimination could result in benefits at the maintenance phase.

*Aspect Feature Envy:* In AspectJ, pointcuts could be defined in aspects and also in classes. If a single aspect uses a class-defined pointcut, it is interesting to move the pointcut from the class to the aspect that uses it.

*Abstract Method Introductions:* Aspects could be used to add state and behavior into existing classes. This is made through the inter-type declaration mechanism. However, the use of this functionality may cause problems when abstract methods are inserted in application classes.

*Large Aspects:* Whenever an aspect tries to deal with more than one concern, it could be divided in as many aspects as there are concerns.

# APPENDIX B

# PUBLICATIONS

## B.1 International Journal

1) Thongmak, M. and Muenchaisri, P. Maintainability Metrics for Aspect-Oriented Software. International Journal of Software Engineering and Knowledge Engineering

## B.2 International Conferences

1) Thongmak, M. and Muenchaisri, P. Checking Violations of the Object-Oriented Design Heuristics. the 17th International Conference on Software Engineering and Data Engineering (SEDE 2008)

# BIOGRAPHY

| | |
|---|---|
| **Name** | Mathupayas Thongmak |
| **Sex** | Female |
| **Date of Birth** | November 8,1979 |
| **Place of Birth** | Bangkok, Thailand |
| **Educations:** | |
| 2007 | Ph.D. in Computer Engineering, Faculty of Engineering, Chulalongkorn University<br>*Funding source*: Faculty of Commerce and Accountancy, Thammasat University |
| 2003 | M.Sc. in Computer Science, Faculty of Engineering, Chulalongkorn University |
| 2000 | B.A. in Management Information Systems, Faculty of Commerce and Accountancy, Thammasat University |
| **Experiences:** | |
| Apr 2002 – Jul 2003 | Department of Computer Engineering, Chulalongkorn University, Thailand<br>*Project:* Object-Oriented Software Metrics (A Collaborative Project between the Stock Exchange of Thailand and Department of Computer Engineering)<br>*Function:* Research Assistant |
| Jul 2004 – Present | Department of Management Information Systems, Faculty of Commerce and Accountancy, Thammasat University<br>*Function:* Lecturer |