Teaching Software Design Techniques in University Courses

Jonathan D. Holmes[1], Kevin R. Parker[1], Bill Davey[2] and Joseph T. Chao[3]
[1]Department of Informatics and Computer Science, Idaho State University, Pocatello, ID, USA
[2]RMIT University, Melbourne, Australia
[3]Department of Computer Science, Bowling Green State University, Bowling Green, OH, USA

Synonyms

Architectural pattern; Design pattern; Development environment; Framework; Software architectural structures; Software architecture; Software design techniques

Definition

Software design techniques focus on reusable software architectural structures such as frameworks and design patterns used to facilitate software development. Software design techniques go well beyond both programming best practices and software design guidelines like cohesion and coupling.

Introduction

Large scale reuse of software architectures and detailed design are made possible through the use of design patterns and frameworks, respectively. Design patterns support reuse of software architecture and design, while frameworks support reuse of detailed design and code (Crawford et al. 2006).

Any academic program responsible for teaching software design concepts must continually seek approaches by which to improve the ways in which students are prepared for the work force. Software design concepts such as design patterns and frameworks are often underemphasized in information systems and computer science curricula. "Textbooks frequently take the approach of demonstrating how to use a particular library or toolkit, spending little time discussing the structure of the program or architectural patterns like MVC" (Hanson and Fossum 2005, p. 120). The IS 2010 Curriculum Guidelines (Topi et al. 2010) state that graduates of undergraduate IS programs should be experts in high-level design of enterprise architecture. "Although the knowledge and skills that IS graduates need have recently moved significantly in the direction toward higher levels of abstraction, individual skills related to design and implementation are still essential for IS graduates" (Topi, p. 20). The Computer Science Curriculum (ACM 2013) guideline includes eight core hours dedicated to design topics such as design patterns, software architecture, structured

© Springer Nature Switzerland AG 2019
A. Tatnall (ed.), *Encyclopedia of Education and Information Technologies*,
https://doi.org/10.1007/978-3-319-60013-0_132-1

design, and object-oriented design, but many programs fail to adequately emphasize each of those topics.

Students are often so focused on learning programming language syntax that they fail to see the big picture of applying that language to solving real problems (Hundley 2008). Development of their problem-solving skills takes a back seat to learning the minutia of a language. Thus, when students take a course in which design topics are covered, instead of focusing solely on those design topics, the instructor must spend valuable time introducing students to the concepts associated with designing a large system.

Review of Design Patterns, Frameworks, and MVC

Design Patterns

One widespread program development paradigm is the use of design patterns. Design patterns represent the embodiment of best-practice programming and support reuse of successful software architecture and design by "capturing the static and dynamic structures and collaborations of successful solutions to problems that arise when building applications in a particular domain. Patterns explicitly capture expert knowledge and design trade-offs and make this expertise more widely available" (Crawford et al. 2006).

The use of design patterns is a fairly recent addition to software development. They were first proposed in an OOPSLA-87 Panel Session by Cunningham and Beck (1988), who in turn attribute the idea to the architectural work of Alexander et al. (1977). The concept went relatively unnoticed until the seminal piece by Gamma et al. (1993), in which they formalize the concept of design patterns and propose a design pattern template.

The concept of design patterns is elaborated upon by Astrachan et al. (1998), Hundley (2008), Reiss (1999), and Wallingford (2002). A design pattern is a documented solution to a specific problem used to guide a designer. It presents the interaction of data and methods in multiple classes, describes the implementation of classes and features needed for the interaction, and offers insight to problems that may arise in the application. Design patterns are reusable solutions that designers can apply, with possible modifications, to an application.

A design pattern is generally defined as a schema comprised of four parts:

- Name – provides a handle and a vocabulary for discussion and use of the pattern.
- Problem – provides a context in which the pattern is applicable.
- Solution – describes the components of the pattern and their interaction, including their responsibilities, relationships, and collaborations.
- Consequences – trade-offs and implications that arise from adapting the solution to the problem.

Design patterns are uniquely named and are written in a consistent format that makes it possible for designers, developers, and others to communicate using a common vocabulary (Khwaja and Alshayeb 2015). They are typically written in design-pattern specification languages that document the abstraction detailed in the design pattern rather than capturing algorithms and data (Khwaja and Alshayeb 2015).

Over the last two decades, software professionals have embraced patterns as a means of recording and sharing expert knowledge for building software. It seems sensible that students should be exposed to this approach to design.

Experience often makes the difference between being a poor and good designer. When determining a solution to a problem, good designers rely on prior experience by relating the problem to those they have previously encountered. Design patterns are the results of formalizing this experienced-based knowledge and document the stereotypical techniques and program structures used by veteran programmers. Each pattern describes a particular scenario with constraints, motivations, and relationships and provides a relatively optimum solution to these problems (Tao et al. 2015). Recent studies by both Hussain et al. (2017) and Gravino and Risi (2017) found that there is a significant relationship between design pattern usage and greater software quality.

Shortly after their seminal conference paper, the book *Design Patterns: Elements of Reusable Object-Oriented Software* was authored by Gamma et al. (1995). The 4 authors, now known as the Gang of Four, presented 23 design patterns for purposes such as object creation, modification, reuse, implementation, and data sharing. These are used as the foundations of all other patterns. These original 23 patterns have since been expanded upon, but there are three basic patterns that are especially useful to understand. They are Singletons, Factories, and Builders.

Singletons

In a singleton design pattern, the instantiation of a class is restricted within a single object, and that object can be shared across an entire application. Singletons are useful when one and only one persistent instance of an object is needed for the entire lifetime of an application. The instance of the object is not created until it is needed. When it is requested, a single object from the class is created and kept inside the class itself. This often is accomplished using a private constructor, ensuring that the only way to create the object is from within the class itself. This makes it possible for the class itself to make sure no more than one object is ever created.

While similar effects can be attained using a static class, static classes can have only static methods, meaning they are purely procedural. Because the singleton is truly object-oriented, it offers several advantages. For example, unlike a static class, a reference to the single instance of a singleton can be passed as a parameter to other methods, and that parameter can be treated like any normal object. Furthermore, a singleton can implement interfaces, inherit from other classes, and permit inheritance. While less competent programmers sometimes use singletons to replace global variables, this is clearly a narrow or incomplete use and often considered poor practice.

A singleton is commonly used when an application has a set of data that does not change often but is frequently referenced. Constantly returning to the data store to retrieve data that has not changed is inefficient and an extensive and unnecessary use of resources. Using a singleton, we can dramatically decrease the number of times we retrieve data that has not changed. A painter’s palette serves as an analogy. It does not make sense to go back to the paint tube each time we need a color. Therefore, we would keep an instantiation of all of the different colored paints we need on the palette and only refresh them when the paint is “dirty.” Examples may include a list of user permission roles, organizational categories (such as gender or age groups), or even a collection of data access layers for multiple data sources.

Factories

In a factory design pattern, objects are created without exposing the creation logic to the client. The goal is to have a single location where all instantiations of a class or series of classes are defined. The objects for these classes cannot be created on their own; they can only be requested from the factory class. Using a constructor that has namespace access modifiers will facilitate this behavior. Factories create not only consistency in how objects are created, but it also creates a single place to go when there are problems. Rather than scouring your entire project to find each line of code in which you created a certain object, you simply go to the factory where it was “assembled” and fix the problem there. It will now propagate that fix through your entire solution. The simplest analogy for a factory is an actual factory. For example, a factory that makes gears will make a specific type of gear. If you want one of those gears, you must request it from that factory, and it will be made for you. If a problem is found with that type of gear, it is as simple as going to the factory and having them fix their error there, and all future gears from that factory will no longer have that problem. Examples for a factory may include a Roster class that is the only place where new employee objects can be created or a Messaging class where all messages sent to users are created.

Builders

You would use the builder design pattern when the construction of an object is complex or has a specific order. Builders are like factories. The main difference being that the builder pattern is used to create complex objects with lots of parts and/or when the order in which the parts are put together matters; however, factories are for less complex construction processes. A simple analogy for a builder could be a construction firm. For example, if you want a house, you might go to the construction contractor and request a house. The house will have lots of parts that they will need to construct because a house has a lot of objects that depend on other objects (inter-object dependencies). The builder's job is to put all of these dependent objects together correctly and in the right order to make this complex object that is a house. The order matters because you cannot put the roof on the house before you have erected the walls, nor can you erect the walls until you have laid the foundation. An example case for a builder may be when you need to build a report that requires particular components of the report to be retrieved before you can retrieve additional related data.

Frameworks

Frameworks facilitate reuse of detailed design and code. "A framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related applications" (Crawford et al. 2006).

A framework defines a family of related applications and contains elements that are common to those applications (Tao 2002). Frameworks provide structure by enforcing naming conventions (directories, files) and rules for constructing a system. They also provide components that aid in the construction of a system. Application developers extend the framework to build custom applications. Thus, a framework is basically a development environment in which programmers can develop applications of all types much easier and faster, including Web applications. A framework may consist of source code libraries, utilities, plug-ins, development models, and a wide range of tools, the purpose of which is to accelerate the application development pace.

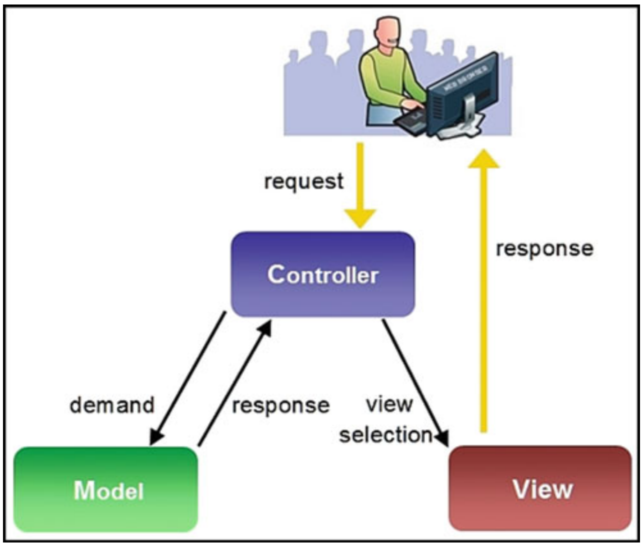
There is a core of recurring themes in most frameworks (Caspersen and Christensen 2008):

- The framework delivers application behavior at a high level of abstraction.
- A framework provides functionality within a well-defined domain, i.e., frameworks address specific domains like graphical user interfaces, gaming, insurance, telecommunication, etc.
- A framework defines the interaction patterns between a set of well-defined components/objects. In order to use the framework, the developers are required to understand these interaction patterns and must program in accordance with them.
- A framework is flexible so that it can be tailored to a concrete context, provided that context lies within the domain of the framework.
- A framework makes possible the reuse of code as well as reuse of design.

MVC

The Model-View-Controller (MVC) pattern is probably the most widely known pattern. Rather than being a design pattern, it is generally agreed that MVC is an architectural pattern. While design patterns typically focus on components of a subsystem and their relationships and collaborations with each other, architectural patterns focus on the larger subsystems of an application along with their relationships and collaborations with each other. (Note that the preceding clarification can be found with multiple attributions on the Internet.)

MVC has proven to be an important architectural pattern for facilitating the development, debugging, and maintenance of systems. The origin and details of the MVC are explained by Tao (2002), Morse and Anderson (2004), and Hanson and Fossum (2005). It was developed for Smalltalk-80 at Xerox PARC in the late 1970s as a means of facilitating the development of systems with graphical user interfaces (GUI).



Teaching Software Design Techniques in University Courses, Fig. 1 MVC patterns (Chao et al. 2013)

MVC was found to be particularly appropriate in applications that provide multiple views of the same data. It has been successfully applied to the development of client-server and web-based systems (Gacimartín et al. 2011; Kasik and Stankus 2010). MVC has been used and adopted to varying degrees in most GUI class libraries and application frameworks, and it has become the central feature of many modern interactive applications (Morse and Anderson 2004).

The key idea behind the MVC pattern involves separating user interfaces from the underlying data represented by the user interface. Thus, the MVC pattern, as shown in Fig. 1, separates the three main components of an application: Models, Views, and Controllers.

- The Model contains the data collected on the client side as well as application methods to access and manipulate the data. This includes both the information represented by the View and the application's business logic that changes this information in response to user interaction. Data models are typically databases or XML files. The Model provides data to the other two components.
- The View is responsible for the visual display of the application. The View component contains scripts that render the View selection for the user and contains all the graphical user interface components that receive input from users. In addition, the View is responsible for displaying output created by the Model for the user.
- The Controller dictates how the application behaves by acting as the primary event handler that responds to events generated by the user through the graphical components in the View. The Controller is the glue between the data models and the views. It connects the presentation (View) and data layer (Model) to each other and coordinates activities between them. Controllers process user input and update the Model and View appropriately. Every controller has a number of actions, and each action will, in general, generate a View for the user. Actions are grouped into controllers based on some criteria, generally a common data model or purpose. Every action is responsible for taking the relevant data, doing whatever controller-specific processing needs to be done, then passing it to the View.

In general, the Model manages application data, the View presents the user interface, and the Controller handles events for the View. Separating the internal data models from the interface views that set or display the data allows models to be represented by a variety of views. Further, the model's algorithmic logic is distinct from the management of the view, thus leading to a simpler program design.

The views must maintain their own state, and there must be communication by which the models update the views or the views update the models. Such communication may be initiated by internal processes or by the user-initiated events. Events may also trigger internal processes that result in updates of the models and/or the views.

Encapsulating the three abstractions (Model, View, and Controller) into separate classes leads to multiple desirable outcomes:

- The same set of business logic (Model) can be used with multiple views to provide different user interfaces for the same underlying application. These interfaces could include web interfaces or traditional GUIs, or could be developed in multiple languages or with different sets of user permissions.
- The impact of user interface changes is minimized. The application's "look" can be drastically altered without modifying data structures and business logic.
- The reusability of domain objects is increased (Tao 2002).
- By separating the model from the view and controller, separate teams of developers can work on each component, either serially or in parallel.
- Automated test harnesses can be used to perform extensive unit testing on the business logic without tedious testing of the GUI.

The MVC pattern has proven to be very useful in industry and also can be effectively used in student projects using either traditional or object-oriented methodologies. Web applications, mobile applications, and other interactive software systems can benefit by being designed with the MVC pattern logic.

Benefits of Patterns and Frameworks

The motivation for using design patterns and frameworks is explained by Ali et al. (2011), Gamma et al. (1993), Vuksanovic and Sudarevic (2011), and Wick (2001).

Little software of any consequence is developed using only a programming language. With the emergence of modern software development methodologies and related frameworks, many developers began to harness their advantages, such as faster development, enhanced security, availability of useful and standardized libraries, simpler organization of work in development teams, and clearer structure of code due to strict conventions and use of patterns that encourage separation of domain logic, user interface, and the data processing model.

Design patterns have become an extremely effective tool with which to solve software development problems because they represent language and application independent solutions to commonly occurring design problems. Developers familiar with design structures can apply them immediately to design problems without having to rediscover them. Less experienced software developers can benefit from design patterns by taking advantage of the lessons and outcomes learned by more experienced developers. Design patterns also facilitate the reuse of successful architectures, because expressing proven techniques as design patterns makes them more readily accessible to other developers.

Most commercial software is developed using a framework by extending and customizing the default, generic functionality that it provides. Thus, frameworks not only help in understanding and constructing real-world applications but also provide additional reusability of both design and implementation. Frameworks offer numerous technical and organizational advantages over classical development methods, such as faster development and cleaner application structure. In addition, programming using frameworks is more comfortable for developers, since frameworks provide for many common programming tasks.

The use of frameworks reduces time spent on code maintenance and future development in many ways:

- Frameworks provide numerous libraries and helpers, making it possible for developers to achieve comparable functionality with less code, so the software is more easily maintainable.

- Software unit testing, defined as a process that includes the performance of test planning, the acquisition of a test set, and the measurement of a test unit against its requirements (IEEE 1986), is facilitated by the framework rather than performed manually.
- Those tasked with software maintenance will require significantly less time deciphering existing code developed via a framework, thanks to the Model-View-Controller pattern and other coding conventions.
- Future improvements, such as protection measures against future hacking attack techniques and yet unknown vulnerabilities, may be addressed by future releases of the framework and thus may only require upgrading the framework to a newer version.

Additional Applications of Patterns and Frameworks

While patterns and frameworks improve software quality, they can also be used to address specific needs. Esakia and McCrickard (2016) discuss the use of patterns and frameworks in the development of mobile apps. Prabhakar et al. (2017) show that the use of various design patterns in the development of various components of data mining applications can improve quality attributes such as reusability, maintainability, extensibility, adaptability, and performance. Colesky et al. (2013) and Van Niekerk and Futcher (2015) explore how design patterns can play a role in introducing security during the groundwork of applications by influencing coding habits of developers. Suphakul and Senivongse (2017) discuss the use of privacy design patterns for the development of privacy-aware applications. A large variety of additional applications for patterns and frameworks can be discovered with an Internet search.

Motivation for Teaching Design Concepts Via Frameworks

One of the most beneficial aspects of design patterns is their educational nature. Creating a design pattern requires a high degree of explicitness. Wallingford (2002, p. 152) explains:

Writing a pattern requires the author to state explicitly the context in which a technique applies and to state explicitly the design concerns and trade-offs involved in implementing a solution. These elements of a pattern provide significant benefits to the reader, who can not only study the technique that defines a solution but also explore when and how to use it.

Astrachan et al. (1998, p. 153) assert that “patterns are an essential programming and pedagogical tool.” Wallingford (2002) agrees, calling for further studies into ways in which to use patterns effectively in teaching.

Christensen and Casperson strongly advocate frameworks in teaching, and provide many convincing arguments for their use topic (Caspersen and Christensen 2008; Christensen 2004; Christensen and Caspersen 2002). The programming context faced by most programmers today is radically different from the one that existed a decade and a half ago. Being a successful developer is no longer a question of being a good programmer, but just as much a question of understanding complex interaction patterns in frameworks and being able to design in accordance with their guidelines. Developing any realistic software requires the ability to identify proper software frameworks to reuse as well as the skill to write the specialized code for the task at hand.

This means that we need to teach new skills to our students in addition to the old ones. Writing even a simple program that has a graphical user interface using a framework is radically different from traditional approaches. The event-driven protocol in a framework requires students to understand the complex interplay between framework code and their own code. Students must be adequately trained to make software reuse their first development option.

While the main motivation for teaching frameworks focuses on the obvious need to teach students about the techniques that govern modern software development, there are additional pedagogical aspects that make frameworks a worthwhile.

- Student motivation: If students must program everything from scratch, then the workload and complexity rule out developing software that in any respect compares to the sophisticated and appealing programs that they are used to from, for example, the Windows platform. However, a framework used in the classroom defines the skeleton of an application that can be customized by an application developer and can provide the "bells and whistles" that makes the effort invested by the student look more appealing or "professional." The "return on investment" is simply greater for the student.
- Setting an example: A well-designed framework can serve as an exemplary use of design patterns by making it clear that design patterns work together and that patterns really define roles.
- Gentler learning curve: Simple frameworks can illustrate basic concepts in framework theory and practice in a gentle way as stepping stone for learning more complex GUI frameworks. It is important that students learn the basic concepts and techniques relating to frameworks.
- Developing is a reuse business: Students must learn that software development entails more than producing code but also involves locating and incorporating reusable assets. Frameworks illustrate many design patterns and reinforce the concept that patterns express roles in a collaboration pattern rather than objects that are to be copied into a design. Frameworks exemplify the reuse of design as well as code, whereas design patterns demonstrate only design reuse. Thus, they serve to strengthen the concept of programming as a process of reuse as well as coding.
- Workforce ready: Frameworks have already made an impact on how industrial software is developed in a cost efficient and reliable way. Graduates should be familiar with the basic concepts and techniques relating to frameworks. Further, students are better prepared for the workforce when exposed to commercial tools.
- Object concepts. Frameworks utilize best practices of object-oriented programming. Requiring students to read and understand parts of frameworks will provide them with important insights in how to structure complex systems in a master-apprentice fashion.
- Responsibility-driven design: An MVC framework is an excellent example of how responsibility-driven design influences the identification of classes and the assignment of responsibilities to those classes. While many students have an initial expectation that the GUI should be integrated with the data since they are so closely related, an MVC framework provides an intuitive and concrete example of how the separation of manipulation and display can result in a powerful and flexible software design (Wick 2001).

Tao et al. (2015) suggest a reasonable set of learning outcomes for teaching patterns and frameworks:

- Gain basic knowledge and understanding of the concept of software architecture, including frameworks and design patterns.
- Become familiar with multiple widely used patterns and get hands-on experience developing simple applications to illustrate learned patterns.
- Develop the ability to consciously apply proper patterns in future software design efforts.

Summary

Several studies have made the case that current curricula neglect design patterns, architectural patterns, and frameworks. While many studies have established the benefits of patterns and frameworks, there is a need for additional research to find ways in which to incorporate these in teaching.

There are many benefits to incorporating patterns and frameworks in the classroom. By introducing students to patterns and frameworks during their education, they gain not only an appreciation for design concepts like separating the user interface from the underlying data and logic but also exposure to commercial software tools that enhances their future marketability.

References

ACM/IEEE-CS Joint Task Force on Computing Curricula (2013) Computer Science Curricula 2013. ACM Press and IEEE Computer Society Press. https://doi.org/10.1145/2534860

Alexander C, Ishikawa S, Silverstein M (1977) A pattern language. Oxford University Press, Oxford

Ali Z, Bolinger J, Herold M, Lynch T, Ramanathan J, Ramnath, R (2011) Teaching object-oriented software design within the context of software frameworks. In: Proceedings of the 41st annual ASEE/IEEE frontiers in education conference, pp 1–5. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.300.7477&rep=rep1&type=pdf

Astrachan O, Berry G, Cox L, Mitchener G (1998) Design patterns: An essential component of CS curricula. In: Proceedings of the twenty-ninth SIGCSE Technical symposium on Computer science education, pp 153–160. http://www.cs.duke.edu/~ola/papers/patterns.pdf

Caspersen ME, Christensen HB (2008) Frameworks in teaching. In: Bennedsen J, Caspersen ME, Kölling M (eds) Reflections on the teaching of programming methods and implementations. Springer-Verlag, Heidelberg, pp 190–205

Chao JC, Parker KR, Davey B (2013) Navigating the framework jungle for teaching web application development. J Issues Inf Sci Inf Technol 10:95–109. https://doi.org/10.28945/1798. http://iisit.org/Vol10/IISITv10p095-109Chao0092.pdf

Christensen HB (2004) Frameworks: putting design patterns into perspective. In: Proceedings of the 9th annual SIGCSE conference on innovation and technology in Computer science education, pp 142–145. http://www.daimi.au.dk/~hbc/publication/iticse2004.pdf

Christensen HB, Caspersen ME (2002) Frameworks in CS1 – a different way of introducing event-driven programming. In: Proceedings of the 7th annual conference on innovation and technology in Computer science education, pp 75–79. http://cs.au.dk/~mec/publications/conference/03%2D%2Diticse2002.pdf

Colesky M, Futcher L, Van Niekerk J (2013) Design patterns for secure software development: demonstrating security through the MVC pattern. In: Proceedings of the 15th annual conference on WWW applications. Cape Town, pp 10–13

Crawford B, Castro C, Monfroy E (2006) Knowledge management in different software development approaches. In: Yakhno T, Neuhold EJ (eds) Advances in information systems. ADVIS 2006. Lecture notes in computer science, vol 4243. Springer, Berlin, pp 304–313

Cunningham W, Beck K (1988) Using a pattern language for programming. Addendum to the Proceedings of OOPSLA’87, ACM SIGPLAN Notices, 23, 5

Esakia A, McCrickard DS (2016) An adaptable model for teaching mobile app development. 2016 IEEE frontiers in education conference. Erie, 2016, pp 1–9. https://doi.org/10.1109/FIE.2016.7757478

Gacimartín C, Hernández J, Larrabeiti D (2011) A middleware architecture for designing TV-based adapted applications for the elderly. In: Jacko J (ed) Human-computer interaction. Design and development approaches, vol 6761. Springer-Verlag, Heidelberg, pp 443–449

Gamma E, Helm R, Johnson R, Vlissides J (1993) Design patterns: abstraction and reuse of object-oriented design. In: Proceedings of the 7th European conference on object-oriented programming, pp 406–431. http://www.cs.pitt.edu/~mock/cs1530/lectures2/ecoop93-patterns.pdf

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional, Indianapolis

Gravino C, Risi M (2017) How the use of design patterns affects the quality of software systems: a preliminary investigation. In: 2017 43rd Euromicro conference on software engineering and advanced applications (SEAA) IEEE, pp 274–277. https://doi.org/10.1109/SEAA.2017.32

Hanson S, Fossum TV (2005) Refactoring model-view-controller. J Comput Sci Coll 21(1):120–129. http://www.cs.uwp.edu/staff/hansen/publications/StopWatch/mvc.ccsc.pdf

Hundley J (2008) A review of using design patterns in CS1. In: Proceedings of the 46th. Annual southeast regional conference on XX, pp 30–33

Hussain S, Keuung J, Khan AA (2017) The effect of Gang-of-Four design patterns usage on design quality attributes. In: 2017 IEEE international conference on software quality, reliability and security (QRS). IEEE, pp 263–273. https://doi.org/10.1109/QRS.2017.37

IEEE (1986) IEEE standard for software unit testing. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=27763

Kasik V, Stankus M (2010) Graphical development system design for creating the FPGA-based applications in biomedicine. In: Bamidis PD, Pallikarakis N (eds) XII Mediterranean conference on medical and biological engineering and computing 2010, vol 29. Springer-Verlag, Heidelberg, pp 879–882

Khwaja S, Alshayeb M (2015) Survey on software design-pattern specification languages. ACM Comput Surv 49(1):1–35. https://doi.org/10.1145/2926966

Morse SF, Anderson CL (2004) Introducing application design and software engineering principles in introductory CS courses: model-view-controller Java application framework. J Comput Sci Coll 20(2):190–201. http://www.wou.edu/~andersc/pubs/CCSC-NW_2004.pdf

Prabhakar NP, Rani D, Hari Narayanan AG, Judy MV (2017) Analyzing the impact of software design patterns in data mining application. In: Dash S, Vijayakumar K, Panigrahi B, Das S (eds) Artificial intelligence and evolutionary computations in engineering systems. advances in intelligent systems and computing, vol 517. Springer, Singapore. https://doi.org/10.1007/978-981-10-3174-8_7

Reiss SP (1999) A practical introduction to software design with C++. Wiley, Hoboken

Suphakul T, Senivongse T (2017) Development of privacy design patterns based on privacy principles and UML. In: 2017 18th IEEE/ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computing (SNPD). IEEE, pp 369–375. https://doi.org/10.1109/SEAA.2017.32

Tao Y (2002) Component- vs. application-level MVC architecture. Frontiers in Education, T2G-7-T2G-10. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.5508&rep=rep1&type=pdf

Tao Y, Liu G, Mottok J, Hackenberg R, Hagel G (2015) Just-in-time-teaching experience in a software design pattern course. In: 2015 IEEE global engineering education conference (EDUCON), pp 915–919. https://doi.org/10.1109/EDUCON.2015.7096082

Topi H et al (2010) IS 2010: Curriculum guidelines for undergraduate degree programs in information systems. Report from the joint IS 2010 curriculum task force. https://www.acm.org/binaries/content/assets/education/curricula-recommendations/is-2010-acm-final.pdf

Van Niekerk J, Futcher L (2015) The use of software design patterns to teach secure software design: an integrated approach. In: Bishop M, Miloslavskaya N, Theocharidou M (eds) Information security education across the curriculum. WISE 2015. IFIP advances in information and communication technology, vol 453. Springer, Cham, pp 75–83. https://doi.org/10.1007/978-3-319-18500-2

Vuksanovic IP, Sudarevic B (2011) Use of web application frameworks in the development of small applications. In: Proceedings of the 34th international convention on information and communication technology, electronics and microelectronics (MIPRO), pp 458–462

Wallingford E (2002) Functional programming patterns and their role in instruction. In: Proceedings of the international conference on functional programming, pp 151–163. http://www.cs.uni.edu/~wallingf/patterns/papers/fdpe2002/fdpe2002.pdf

Wick MR (2001) Kaleidoscope: using design patterns in CS1. In: Proceedings of the 32nd SIGCSE technical symposium on computer science education, pp 258–262