Introduction to Software Design

1

The objective of this chapter is to provide context for what is and what is not software design, to introduce the notion of abstraction, and to position software design within a software development process.

1.1 Preconditions

The following should be true prior to starting this chapter.

- You have experience developing and testing program code.

1.2 Concepts and Context

This section presents an introduction to software design by answering the following questions.

1. What is software design?
2. What is not software design?
3. What are some approaches for creating a software design?
4. What are some examples to help me better understand the concept of abstraction?
5. How does software design fit into a software development process?
6. How does the selection of a software development process affect your software design approach?

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_1

1.2.1 What is Software Design?

There are many good answers to this question. The answers below are from four perspectives.

1.2.1.1 Software Design is a Process that, When Performed, Translates Requirements into a Design. A Software Design Process is a Bridge Between What the Software Needs to do (e.g., Requirements) and the Software Implementation (e.g., Code)

This answer explains software design from the perspective of a software development process (SDP). An SDP is a description of the steps to be performed in order to develop software.

At this point in your learning, you've likely been given programming assignments that describe what the code should do. Your task was to write the code that meets the assignment description. In essence, the instructor completed many of the SDP steps for you, leaving you with the task of writing and testing your code.

There are many SDPs, each having a unique list of steps to be performed, and entire books have been written to explain a single SDP. SDPs also vary in the types of artifacts that are created, where an artifact is an output produced by performing one or more SDP steps. The SDP topic is large and complex, and there is a significant debate about which type of SDP is better to use. The following paragraph provides a simplified description of some SDPs.

Many SDP descriptions will group steps together that are logically related to each other and give this group of steps a name. These SDPs will likely have a group that contains planning steps, analysis steps, design steps, coding steps, and testing steps. These groups are often called phases. Thus, many SDP descriptions include a plan phase, an analysis phase, a design phase, a code phase, and a test phase.

This first answer to the question *what is software design?* can be explained using these SDP phases. Before software code can be written, the developers need to know what the software needs to do. These needs are often called requirements, and requirements are identified by performing the steps in the analysis phase. Once we know the requirements, we can begin to formulate how the software code should be constructed. These formulations, called software design models, would be created by performing the steps in the design phase. Once we have design models, we can perform the steps in the code phase to translate the design models into code.[1]

[1] As mentioned, there is much debate about which SDP is best to use. Some researchers and practitioners promote the agile SDPs as being better than the more traditional approaches. I'll address this debate later in this chapter (see Sect. 1.2.5), focusing specifically on how the different types of SDPs address the need for software design. This book is focused on learning how to design software; this focus lends itself to some of the more traditional SDPs that have fairly clear descriptions for the SDP phases just described.

1.2.1.2 Software Design is a Description of the Structure and Behavior of Software at a Higher Level of Abstraction Than the Code.

This answer explains software design from the perspective of a designer. A software designer wants to describe software elements found in the solution and how these elements are associated with each other. Descriptions of software elements within a design document (i.e., artifact) are used to depict the solution at various levels of abstraction. A high-level design artifact may describe the major components and how these are associated with each other. A lower level design artifact may describe the modules/classes and how these more detailed design elements are associated with each other.

When you write software code, you are focusing on building functions (in procedural code) or methods (in object-oriented code) that are responsible for certain processing. In procedural code, functions are grouped together to create a module, and many modules may be combined to solve the problem. In object-oriented code, methods are grouped together to create a class, and many classes may be combined to solve the problem. Thus, describing the structure of a software design depends on the type of programming paradigm[2] used in the code.

- Procedural code is represented by having functions within a module and by having these modules interact with each other to solve the problem. A software design artifact that describes the *structure* of a procedural solution would need to accurately represent the notion that functions are found within modules, that modules may be combined to represent a component, and that components may be combined to represent the entire software application.
- Object-oriented code is represented by having methods within a class and by having these classes interact with each other to solve the problem. A software design artifact that describes the *structure* of an object-oriented solution would need to accurately represent the notion that methods are found within classes, that classes may be combined to represent a package, and that packages may be combined to represent the entire software application.

Software *behavior* is represented by the detailed interactions between modules/functions (in procedural code) or between classes/methods (in object-oriented code). These behaviors manifest themselves in code in three different ways.

- Software code uses three types of language statements to represent behavior. These statement types are sequence, selection, and iteration. A software design artifact describing behavior may choose to document the algorithm being implemented in the code. An algorithm may be documented using a natural language, pseudo-code, or a diagram to represent logic expressed using sequence, selection, and iteration.

[2]The two programming paradigms discussed in this book are procedural (aka imperative) and object-oriented. Procedural code may be written in languages like C, COBOL, C++, and Python. Object-oriented code may be written in languages like C++, Java, and Python. Other language paradigms include functional (e.g., lisp, scheme), logical (e.g., prolog), and declarative (e.g., HTML, SQL).

- Software code uses function or method calls, including passing parameters and optionally returning a value/object, to represent behavior. A software design artifact that describes behavior may choose to document these function/method calls.
- Software code uses variables to represent the state of the software. This notion of program state is another form of software behavior that may be described in a software design artifact.

At this point in your learning, you may have developed some program code that had a handful of functions in a module or a handful of methods in a class. You may have developed some code that required a couple of modules or classes with each containing a bunch of functions or methods. In these cases, the amount of code you needed to solve the problem was not excessive—you were able to understand the structure and behavior of your solution by reading your code.

However, when your code is orders of magnitude larger, it becomes much harder to understand the structure and behavior of a solution by reading the code. An inexperienced programmer may find it very difficult to comprehend the structure and behavior of a solution with thousands of source lines of code (SLOC), whereas a solution of a few hundred SLOC is relatively easy for them to read and understand. An experienced programmer may be able to understand a solution of a few thousand SLOC by reading the code, but a solution of tens of thousands (or larger) of SLOC would be difficult for any programmer to read and understand in a short amount of time.

The ability to quickly understand the structure and behavior of a solution is one reason why software design is important to learn. A good software designer will accurately represent the implementation code while allowing someone to understand the solution structure and behavior in far less time than having to read the code to get their understanding. A software designer needs to develop the ability to create abstractions that accurately represent the code.

1.2.1.3 Software Design is a Collection of Artifacts that Describe the Architecture, Data, Interfaces, and Components of the Software

The brief description of SDPs found above included using the term artifacts to refer to items that are produced when performing steps in a SDP. Software design artifacts should include a description of the four design categories [1] described in Table 1.1.

A software design artifact may combine text and graphical notations (i.e., models) to describe the structure and/or behavior associated with one of the four design categories. Examples of this type of design artifact are class diagrams, entity–relationship diagrams, data flow diagrams, state machine diagrams, flowcharts, data dictionaries, and pseudo-code.

A software design artifact may be a single document that contains a section for each design category—architecture, data, interfaces, and algorithms. This type of software design artifact consolidates various descriptions of the structure and/or behavior into one coherent document. Having a single design document is simply a

Table 1.1 Four Design Categories

Architecture	A high-level description of the design elements found within a system
Data	A description of the logical and physical representations of the data structures used by the system. These data structures may be used for persistent or volatile storage of data. An example of persistent data storage is a database stored on a hard drive. An example of volatile data storage is a data structure created and used during program execution. When the program execution ends, the memory-based data structure is destroyed, causing any data stored in the data structure to be lost
Interfaces	A description of the human–computer interface, the interfaces between the high-level design elements (described in the architecture), and any interfaces to external systems
Components	A description of the significant or unique processing steps contained within a high-level design element

way to organize the many individual design artifacts into a coherent whole. Chapter 34 presents a template for a single document to contain an entire software design.

1.2.1.4 Software Design is a High-Level Description of the Knowledge Represented by the Code

Many researchers consider the process of developing software to be a knowledge acquisition activity [2–4]. Writing code requires knowledge of the application domain and programming language. The code represents the storage of knowledge about the application domain. Since a software design is an abstraction of the code, software design artifacts also represent knowledge about the application domain. In addition, these artifacts represent knowledge about the software development process being used.

1.2.2 What is Not Software Design?

While it is critical to know what software design is, it is just as important to know what software design is not.

1.2.2.1 Software Design is Not Software Analysis

In software analysis, we focus on understanding *what* the software must do. Software analysis gathers and organizes information that describes the needs (i.e., requirements) that the software must fulfill. In contrast, software design focuses on the structures and behaviors that explain *how* the software satisfies the requirements. Software design applies technical and application domain knowledge to translate the needs into a design.

1.2.2.2 Software Design is Not Program Design

In program design, we focus on the coding details of our solution. Things like good variable names, good method/function names, clean method/function signatures, and the reuse of methods/functions are things we think about when doing program design. In software design, we focus on a higher level of abstraction.

1.2.2.3 Software Design is Not Programming

When programming, we focus on the syntax and semantics of a particular programming language. We use this language and its features to construct code. In software design, we focus on the syntax and semantics of modeling techniques and our natural language to communicate in graphical and written/verbal forms.

1.2.3 What are Some Approaches for Creating a Software Design?

Seven approaches for developing a software design are described in this section, including the advantages and challenges of each approach. The approaches described below are not intended to be a complete list of design approaches. Rather, these seven approaches are used to highlight the range of possible ways a design may be constructed.

1.2.3.1 Top-Down Design

A top-down approach develops a design through a process of subdivision. You start with a high-level context of the system and subdivide this context into logically connected design elements. You may then take each of these design elements and subdivide again, and so on. At some point, each subdivided design element is small enough to be well understood.

The top-down approach is also called *stepwise refinement* or *decompositional*. You produce a high-level design (e.g., identify components of the system). You then refine/decompose this design into a more detailed design (e.g., identify subcomponents for each component). You then refine/decompose this design into a more detailed design (e.g., identify design elements within each subcomponent). At some point, you've refined/decomposed into design elements that are small enough to be correctly implemented via code.

Doing stepwise refinement (i.e., taking a general concept and breaking it down into smaller concepts) is a natural process for us to do; it is a natural way for us to think. In addition, there are many examples in the physical world that allow us to see a device in terms of an entire system, its major components, and its detailed parts (e.g., automotive designs, house designs).

Doing a top-down design does have its challenges.

- Decisions made at a higher level of design will directly influence the lower levels of design. For example, perhaps we've decided that our high-level design should

consist of three components. As we begin to further divide each component, we identify a design element that does not fit nicely within one of the three components. How do we deal with this? Do we need to explore design options as fully as possible at each stage of decomposition?
- Knowing when to stop decomposing is also difficult to judge. How small should each decomposed design element be before we stop? How much detail do we need in our design? How much design detail is too much?
- Finally, decomposing may result in duplicate design elements in two distinct parts of a system. How do we avoid the possibility of duplicate design elements?

1.2.3.2 Bottom-Up Design

A bottom-up approach develops a design through a process of building up from basic elements to the entire system. You start with basic elements and combine these into logically connected subsystems (or subcomponents). You then take each subsystem and combine again, and so on. At some point, the entire system is described as a collection of basic elements.

The bottom-up approach is also called *compositional*. You produce specific design elements (e.g., data structures, methods/functions, classes). You then combine/compose these into a more general design (e.g., physical data model, class diagrams). You then combine/compose these into even more general design elements (e.g., logical data model, package diagram).

Following a bottom-up approach allows us to immediately start to code, thinking about design only when we need to combine individual code elements into a larger solution. Since most software developers start their career by learning to write code, this is a more comfortable starting point. Typically, someone has written code longer than they've done any other software development skill.

Doing a bottom-up design does have its challenges.

- Combining small design elements into a larger design may result in more rework of existing code/design. In fact, the term *refactoring* describes this exact situation. Doing lots of refactoring each time you add code may suggest that a fundamental design flaw exists in your higher level design.
- How do we identify basic design elements from the problem description? Some code/design elements will be apparent by reading the problem statement or by your prior knowledge about the problem domain. However, there are likely code/design elements that are necessary just to make the design work. How and when is this *glue* code identified?
- Identifying and combining basic design elements may be a less intuitive approach when compared to thinking top-down (i.e., hierarchically). Is learning to design using a bottom-up approach harder to learn and apply?

1.2.3.3 Process-Oriented Design

A process-oriented approach develops a design by focusing on the processes/functions being automated. Here, the terms process and function refer to the activities being performed within an organization. These activities are candidates for automation. Design models created in a process-oriented approach emphasize how processing steps are organized (i.e., performed sequentially or concurrently) to meet the needs of the organization. While this approach focuses on process/function, many of the process-oriented design models will show how information/data is being created or manipulated by the activities being automated.

Performing a process-oriented design is beneficial for organizations that have consistent (and documented) policies and procedures for the work being performed. This allows someone to translate the policies and procedures into automated steps. The terms *work flow* and *business process reengineering* are often used to describe this approach.

Following a process-oriented approach to designing software may place constraints on your design, limiting your ability to innovate. This may happen if you focus on automating existing processes without keeping an eye on making the process more efficient. Another possible weakness is that data tends to be included in a design only when viewed from the perspective of the processes that create or use the data. This could result in data structures that are designed based on work flow without regard to how the data is related to each other.

1.2.3.4 Data-Driven Design

A data-driven approach develops a design by focusing on the information/data being automated. Design models created in a data-driven approach emphasize how information/data are organized (i.e., relationships between data elements) to meet the needs of the organization. While information/data is the focus, many of the data-oriented design models will show how processing steps use the information/data.

Doing a data-driven design approach allows a designer to develop the data structures first, and then apply these structures within the appropriate processing elements identified in the design. Since software exists to act upon data (i.e., can you think of a software program that does not use any data?), thinking about data first and processing second may lead to a well-designed system.

Creating a data-driven design may limit your ability to streamline the processing involved in using the data. This is because the design is focusing on data relationships instead of processing steps. The weaknesses of the data-driven approach is analogous to the process-oriented approach. Both of these design approaches can result in weaker designs since their focus is on only one aspect of software.

1.2.3.5 Object-Oriented Design

An object-oriented approach develops a design by focusing on both process and data. In this approach, processing steps and information/data are combined into design elements called classes. A class encapsulates the processing and information into a single software element.

Creating an object-oriented design has the benefits associated with the process-oriented and data-driven approaches, while addressing many of the weaknesses of these two approaches. However, the notion of a class as a software abstraction is not an easy thing to learn. It often takes years to develop the experience necessary to consistently develop good object-oriented software designs.

1.2.3.6 Structured Design

A structured approach develops a design by identifying modules that are often arranged in a hierarchy. Thus, this has much in common with the top-down design approach previously discussed. A structured design may focus on process, data, or both.

1.2.3.7 Hybrid Design Approaches

Often, a design phase in an SDP does not strictly adhere to just one of the above approaches. For example, a design phase may describe a series of steps to be performed where some of the steps ask a designer to think about the entire system, with the goal of establishing some high-level design structures. These steps have the designer following a top-down approach. Other steps in the design phase may ask a designer to think about specific details of the system, with the goal of establishing some detailed design elements. These steps have the designer following a bottom-up approach. The use of both top-down and bottom-up approaches is a natural way to develop a software design. It allows you to think about the entire system while also focusing on those details that may be critical to the success of the software system.

Similarly, a design phase in an SDP may include steps describing the use of process-oriented, data-driven, and object-oriented approaches. For example, a design phase may include steps describing the: development of design models using the Unified Modeling Language (UML) (i.e., object-oriented approach); development of logical data models using entity–relationship notation (i.e., data-driven approach); and development of work flow diagrams representing activities to be automated (i.e., process-oriented approach).

As stated above, the design approaches just described represent some of the common ways a software design is created. Other design approaches included in an SDP may involve doing a function-oriented design or doing an event-driven design. A function-oriented design focuses on functions as the primary software abstraction. Functions can be described at a high-level of abstraction and then be subdivided into smaller functions that may be directly translated into code. An event-driven design focuses on how software should react to external stimuli. Certain types of software, including embedded software and graphical user interfaces, use this design approach. The design identifies the types of events to occur along with how the software should react to each type of event.

1.2.4 What is Abstraction?

The fourth definition of abstraction found at [5] fits the software design notion perfectly: "The act of comparing commonality between distinct objects and organizing using those similarities; the act of generalizing characteristics; the product of said generalization."

1.2.4.1 What Are Some Examples to Help me Better Understand the Concept of Abstraction?

An example of abstraction can be found in the brief SDP explanations given above. I can describe an SDP to you by talking about all of the detailed steps that must be performed. For example, when talking to another software developer I can say "I'm currently interviewing users to understand their needs" or "I'm currently modeling the manual steps we plan to automate." The software developer hearing these statements may not remember the specific details, but they will likely know that I am doing analysis. Alternatively, I can describe an SDP to you by talking about the phases that must be performed. When talking to another software developer, I can say "I'm currently doing analysis." The software developer hearing this would understand that I am currently learning about what the software must do. This person could then ask questions about the types of analysis steps we are performing (if they care to know about the details).

In software design, abstraction is about identifying broader themes from the specific requirements. For example, if the requirements indicate the need for a baseball, softball, and kickball, then we should see an opportunity to generalize these into the need for a ball that is a sphere.

As mentioned above, the term program state is used to describe the execution state of a software program. A program state is represented by the set of variables and their respective values at a given point in time. For example, an assignment statement (e.g., varName = expression) is used to alter a program state. Software designs should show the *significant* changes in state within the program or system being designed. For example, a human resources and benefits system would need to store a change in state when an employee retires. This would likely be a significant state change as the benefits available for an employee will likely be different from the benefits available to a retiree. This form of abstraction will ignore certain details, either because the detail is considered insignificant or because it is not associated with the design element being described.

Another example of abstraction is in describing programming languages. All programming languages allow a person to develop a computational model that may be executed on a computing device. The level of abstraction for a programming language is based on how close the programming language is to the hardware capabilities of the device.

In Fig. 1.1, machine language is at a low level of abstraction; this language is closest to the computing hardware. That is, the instructions found in machine language have a direct relationship to the capabilities of the hardware. In contrast, Python is at

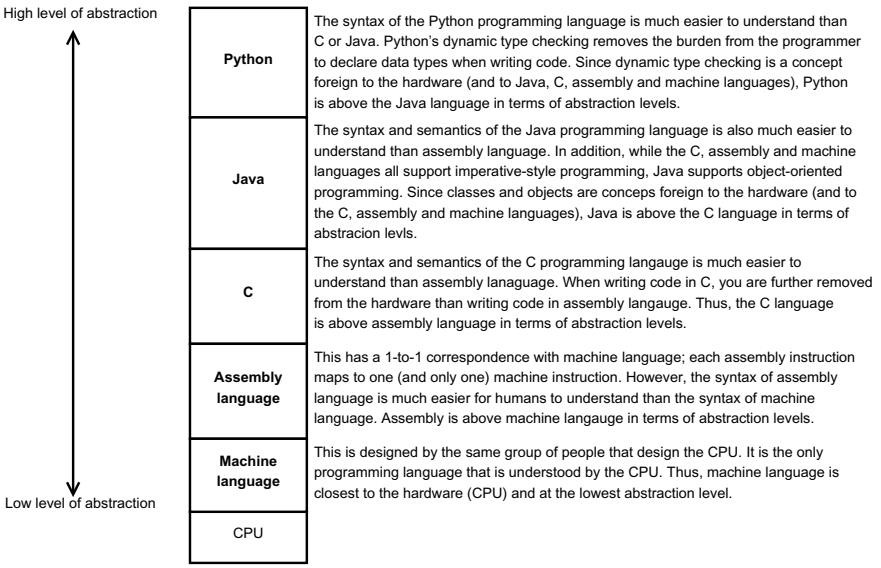

Fig. 1.1 Levels of abstraction—programming languages

a high level of abstraction; this language is further removed from the hardware than the other languages listed in the figure. While Python statements are ultimately executed on hardware, this execution occurs by using an interpreter and other software components that provide translations of Python statements into an executable form that is understood by the hardware.

1.2.5 Software Design Within a Software Development Process

This section provides context by describing software design within the broader topic of software development processes.

1.2.5.1 How Does Software Design Fit into a Software Development Process?

A software development process (SDP) is a series of steps describing the tasks to perform to create a software product. There are many different types of SDPs, each describing a unique approach to developing software. While each type of SDP describes a different approach, all SDPs have a few key things in common. Figure 1.2 shows three common steps performed in most types of SDPs. Each type of SDP describes an approach to gather information about what the software product needs to do. This information is often called requirements. The requirements are then used to produce a design, which is then implemented to produce the software product.

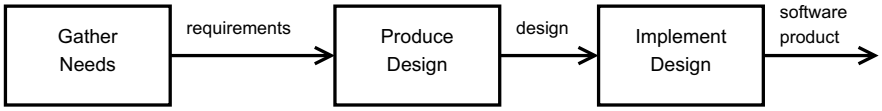

Fig. 1.2 Common steps in SDPs

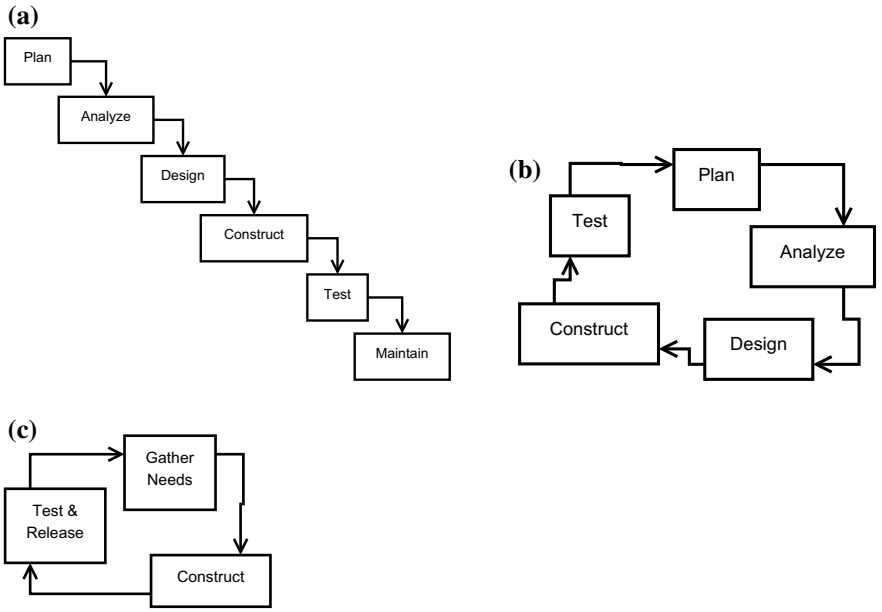

Fig. 1.3 (**a**)Waterfall, (**b**) Incremental and (**c**) Agile

Figure 1.2 shows how the process of software design acts as a bridge between gathering requirements and writing code. A software design process translates the requirements, often written in a native language (e.g., English), into a technical description that is better understood by people responsible for writing code.

1.2.5.2 How Does the Selection of a Software Development Process Affect Your Software Design Approach?

Figure 1.3a, b, and c shows three generic SDPs for a waterfall, incremental, and agile approach, respectively. These three figures are used to illustrate the similarities and differences between these approaches and, in particular, how software design fits within these approaches.

A waterfall SDP is the oldest approach for developing a software product. Its two primary characteristics are: (1) each step/phase produces one or more software documents input into the next step; and, (2) each step/phase is performed only once. Thus, a software product is produced through the sequential completion of the steps described in a waterfall SDP.

Software design in a waterfall SDP adheres to the common steps in SDPs as shown in Fig. 1.2. The *Design* step would receive as input a requirements document produced by the *Analyze* step. The Design step would then produce one or more software design documents, which are input into the *Construct* step. Some waterfall approaches will have two distinct design steps, a high-level design and a detailed design. The high-level design phase would focus on the entire system and its components. The detailed design would decompose the components into subcomponents and smaller design elements, with the goal of having each design element be detailed enough to allow for construction.

An incremental SDP is a commonly used approach for developing a software product. Its two primary characteristics are: (1) each step/phase produces one or more software documents that are then input into the next step; and, (2) the steps/phases are performed many times in a cycle, until the entire software product has been created. Thus, a software product is produced through the iterative completion of the steps described in an incremental SDP.

Software design in an incremental SDP adheres to the common steps in SDPs as shown in Fig. 1.2. The *Design* step would receive as input a requirements document produced by the *Analyze* step. The Design step would then produce one or more software design documents, which are input into the *Construct* step. The significant difference between a waterfall and incremental SDP is the contents of these software documents. In a waterfall SDP, each document produced by a step contains all of the pertinent information related to the software product being produced. In contrast, the documents produced by a step in an incremental SDP contains only the information needed for this particular cycle (or iteration) of the steps. As the iterations continue, each document is updated to reflect additional information needed to incrementally update the software product. Thus, an incremental SDP produces a software design in an evolutionary way, gradually adding design knowledge as more becomes known about the application domain.

An agile SDP is a relatively new approach for developing a software product. Its four primary characteristics are: (1) gather needs through human interactions; (2) code is the only form of technical documentation; (3) incremental development; and, (4) small increments with frequent releases of the software product.

Software design in an agile SDP does *not* adhere to the common steps in SDPs as shown in Fig. 1.2. In an agile SDP, no software design documents are typically created. Instead, the agile manifesto promotes the code as the only form of technical documentation that is needed. Agile software development is an example of following a bottom-up design approach. An agile project will typically experience lots of refactoring[3] as more becomes known about the capabilities being added to the

[3] Refactoring is the process of altering the structure of code to address changes in domain knowledge and/or technical details that were not known at the time the code was first developed. Examples of refactoring in an object-oriented language include splitting a class into many classes, combining many classes into one, splitting a method into many methods, combing many methods into one, and moving methods to a different class. Refactoring in a structured language includes splitting and combining modules and functions in ways analogous to OOP languages.

software product. From my perspective, a controversial aspect of the agile manifesto is its reliance on code as the only form of technical documentation. For projects with a relatively small scope and no more than 2–3 project team members, relying only on the code to document your design may result in a successful project. However, as project scope and/or project team size increases, the need to create design documents to effectively communicate important technical information will be important for the successful completion of the project.

1.3 Post-conditions

The following should have been learned after completing this chapter.

- Software design is a process. When the steps in a software design process are performed, software artifacts are produced to describe how the software will fulfill the requirements.
- Software designs are more beneficial as the size of the software increases.
- A software design describes the structure and behavior of the software. These structures and behaviors represent a higher level of abstraction when compared to the code.
- Software design artifacts should collectively describe the architecture, data, interfaces, and components of the software.
- Top-down is a software design approach that subdivides the problem domain into smaller design elements. For example, the domain is split into components, then the components are split into subcomponents, and so on, until the design elements are small enough to be well understood.
- Bottom-up is a software design approach that creates basic design elements and then combines these into larger design elements. Creating and combining basic design elements continues until all of the requirements have been fulfilled.
- Structured design artifacts represent abstractions of code written using modules and functions.
- Object-oriented design artifacts represent abstractions of code written using classes and methods.
- The following terms and acronyms were introduced in this chapter.

Design category	Identifies the types of design knowledge that are collected and stored in an artifact. There are four categories of design knowledge: 1. Architecture 2. Data 3. Interfaces 4. Components
SDP	Software development process
SLOC	Source lines of code

Exercises

Discussion Questions

1. From a code perspective, what are some examples of a structure? Of a behavior?
2. A *flowchart* or *Nassi–Shneiderman diagram* may be used to model the code using graphic notations that represent sequence, selection, and iteration statements. Typically, these two modeling techniques are used to graphically represent the individual statements/steps found in your code/algorithm. Using the Wikipedia pages as your source, describe how these two modeling techniques could be used to represent more abstract design models instead of the detailed processing steps they were originally intended to convey.
3. According to Pressman (2005), a design should contain artifacts that describe four design categories: architecture, data, interfaces, and components. Based on Table 1.1 and your programming experience, give an example of something you've done in your code to represent:

    a. The architecture of your solution.
    b. The data of your solution.
    c. An interface of your solution.
    d. A component of your solution.

4. An SDP provides guidance to a software development team by describing the types of steps to perform and the types of artifacts to be produced. Can you think of a situation where it may not be necessary to adhere to an SDP when developing software? For example, can you think of a situation where you may want to develop software in an ad hoc manner?

References

1. Pressman RS (2005) Software engineering: a practitioner's approach, 6th edn. McGraw-Hill, New York
2. Abbas N, Andersson J, Weyns D (2011) Knowledge evolution in autonomic software product lines. In: Proceedings of the 15th international software product line conference (SPLC '11), Munich, Germany, vol 2, no 36
3. Armour PG (2000) The business of software: the case for a new business model. Commun ACM 43(8)
4. Armour PG (2004) Beware of counting LOC. Commun ACM 47(3)
5. Wiktionary.org: abstraction (2019) In: Wiktionary The free dictionary. Wikimedia Foundation. https://en.wiktionary.org/wiki/abstraction. Accessed 6 Jan 2019