Software Development and Professional Practice

Lecture 1 : Introduction to Software Development

For

Faculty of Computers and Information

Assiut University

By

Dr. Mostafa Salem

Assistant Professor, FCI-Assiut University

https://mostafasalem.netlify.app

Introduction

- What is Software Development?
  - **Software development** is the process of taking a set of requirements from a user (a problem statement), analyzing them, designing a solution to the problem, and then implementing that solution on a computer.
- Well, isn't that programming? Well, no.
  - Programming is really the implementation part, or possibly the design and implementation part, of software development. Programming is central to software development, but it's not the whole thing.
- Well, then, isn't it software engineering?
  - Again, **no**. Software engineering also involves a process and includes software development, but it also includes the entire management side of creating a computer program that people will use.

Introduction (Cont.)

- Software engineering includes:
  - project management, configuration management, scheduling and estimation, baseline building and scheduling, managing people, and several other things.
- Software development is the fun part of software engineering.
- So, software development is a narrowing of the focus of software engineering to just that part concerned with the creation of the actual software. And it's a broadening of the focus of programming to include analysis, design and release issues.

What We're Doing

- Developing software is hard. Learning how to develop software correctly, efficiently, and beautifully is also hard.
- It's a skill you need to pick up and practice – a lot.
- Several ways to learn software development:
  - reading excellent designs, reading a lot of code, writing a lot of code, and thinking deeply about how you approach a problem and design a solution for it.
- Reading a lot of code, especially really beautiful and efficient code, gives you lots of good examples about how to think about problems and approach their solution in a particular style.

What We're Doing

- Writing a lot of code lets you experiment with the styles and examples you've seen in your reading.
- Thinking deeply about problem solving lets you examine how you work and how you do design, and lets you extract from your labors those patterns that work for you; it makes your programming more intentional.

So, How to Develop Software?

- Despite the fact that software development is only part of software engineering, software development is the heart of every software project.
- After all, at the end of the day what you deliver to the user is working code.

So, How to Develop Software?

- In order to do software development well you need the following:
  - **A small, well integrated team.**
    - Small teams have fewer lines of communication than larger ones.
    - It’s easier to get to know your teammates on a small team.
    - You can get to know your teammates’ strengths and weaknesses, who knows what, and who is the “go to guy” for particular problems or particular tools.
    - Well-integrated teams have usually worked on several projects together. Keeping a team together across several projects is a major job of the team’s manager.
    - Well-integrated teams are more productive, they are better at holding to a schedule, and they produce code with fewer defects at release.

So, How to Develop Software?

- In order to do software development well you need the following:
  - **Good communication among team members.**
    - Teams that are co-located are better at communicating and communicate more than teams that are distributed geographically (even if they're just on different floors or wings of a building).
  - **Good communication between the team and the customer.**
    - Communication with the customer is essential to controlling requirements and requirements churn during a project.
    - On-site or close-by customers allow for constant interaction with the development team.
    - Customers can give immediate feedback on new releases and be involved in creating system and acceptance tests for the product.
    - Ex. The Extreme Programming agile development methodology

So, How to Develop Software?

- In order to do software development well you need the following:
  - **A process that everyone buys into.**
    - Every project, no matter how big or small, follows a process.
    - Larger projects and larger teams tend to be more plan-driven and follow processes with more rules and documentation required.
    - Larger projects do require more coordination and tighter controls on communication and configuration management.
    - **Smaller projects** and smaller teams will tend to follow more agile development processes, with more flexibility and less documentation required.
    - This certainly doesn't mean there is no process in an agile project, it just means you do what makes sense for the project you're writing so that you can satisfy all the requirements, meet the schedule, and produce a quality product.

So, How to Develop Software?

- In order to do software development well you need the following:
  - **The ability to be flexible about that process.**
    - No project ever proceeds as you think it will on the first day.
    - Requirements change, people come and go, tools don't work out, and so on.
    - This point is all about handling risk in your project. If you identify risks, plan to mitigate them, and then have a contingency plan to address the event where the risk actually occurs, you'll be in much better shape. Chapter 4 talks about requirements and risk.

So, How to Develop Software?

- In order to do software development well you need the following:
  - **A plan that everyone buys into.**
    - You shouldn't launch a software development project without a plan.
    - The project plan encapsulates what you're going to do to implement your project.
    - It talks about process, risks, resources, tools, requirements management, estimates, schedules, configuration management, and delivery.
    - It doesn't have to be long and it doesn't need to contain all the minute details of the everyday life of the project, but everyone on the team needs to have input into it, they need to understand it, and they need to agree with it. Chapter 3 talks for more details on project plans.

So, How to Develop Software?

- In order to do software development well you need the following:
  - **To know where you are at all times.**
    - It's that communication thing again.
    - Most projects have regular status meetings so that the developers can "sync up" on their current status and get a feel for the status of the entire project.
    - This works very well for smaller teams (say, up to about 20 developers).
    - Many small teams will have daily meetings to sync up at the beginning of each day.
    - Different process models handle this "spot" meeting differently.
    - Many plan-driven models don't require these meetings, depending on the team managers to communicate with each other.
    - Agile processes often require daily meetings to improve communications among team members and to create a sense of camaraderie within the team.

So, How to Develop Software?

- In order to do software development well you need the following:
  - **To be brave enough to say, “hey, we’re behind!”**
    - Software developers are an optimistic bunch, generally, and it shows in their estimates of work. “Sure, I can get that done in a week!”
    - At some point you’ll be behind. And the best thing to do about it is to tell your manager right away.
    - The earlier you figure out you’re behind, the more options you have.
      - lengthening the schedule
      - moving some requirements to a future release
      - getting additional help, etc.
    - The important part is to keep your manager informed.

So, How to Develop Software?

- In order to do software development well you need the following:
  - **The right tools and the right practices for this project.**
    - One of the best things about software development is that every project is different. Even if you're doing version 8.0 of an existing product, things change.
    - The three most important factors in choosing tools are the application type you are writing, the target platform, and the development platform. You usually can't do anything about any of these three things, so once you know what they are, you can pick tools that improve your productivity.
    - A fourth and nearly as important factor in tool choice is the composition and experience of the development team.
  - **To realize that you don't know everything you need to know at the beginning of the project.**

Conclusion

- Software development is the heart of every software project, and it is the heart of software engineering.
- Its objective is to deliver excellent, defect-free code to users on time and within budget –all in the face of constantly changing requirements.
- That makes development a particularly hard job to do. But finding a solution to a difficult problem and getting your code to work correctly is just about **the coolest feeling in the world**.

Software Development and Professional Practice

Lecture 2: Process Life Cycle Models

For
Faculty of Computers and Information
Assiut University

By
Dr. Mostafa Salem
Assistant Professor, FCI-Assiut University
https://mostafasalem.netlify.app

Introduction

- Every program has a life cycle. It doesn't matter how large or small the program is, or how many people are working on the project – all programs go through the same steps:

1- Conception

2- Requirements gathering/exploration/modeling

3- Design 4- Coding and debugging

5- Testing 6- Release

7- Maintenance/software evolution 8- Retirement

One's program may compress some of these steps, or combine two or more steps into a single piece of work, but all programs go through all steps.

Introduction

- Every life cycle model, however, is a variation on two fundamental types:
  - In the first type, the project team will generally do a complete life cycle – at least steps 2 through 7 – before they go back and start on the next version of the product.
  - In the second type, which is more prevalent these days, the project team will generally do a partial life cycle – usually steps 3 through 5 – and iterate through those steps several times before proceeding to the release step.

Introduction

- These days the management of software development projects generally fall into two different types Traditional plan-driven models, and the newer agile development models.
- **In the plan-driven models:**
  - the process tends to be stricter in terms of process steps and when releases happen.
  - Plan-driven models have more clearly defined phases, and more requirements for sign-off on completion of a phase before moving on to the next phase.
  - Plan-driven models require **more documentation** of each phase and verification of completion of each work product.
  - These tend to work well for **government contracts** for new software with well-defined deliverables.
- The agile models are inherently incremental, and make the assumption that small, frequent releases produce a more robust product than larger, less frequent ones.
- Phases in agile models tend to **blur** together more than in plan-driven models, and there tends to be less documentation of work products required, the basic idea being that code is what is being produced and so documentation efforts should focus there.

Code and Fix

- It isn’t really a model at all.
- It is what most of us do when we’re working on small projects by ourselves, or maybe with a single partner.
- In this model there are **no** formal requirements, **no** required documentation, **no** quality assurance or formal testing, and release is haphazard at best.
- Don’t even think about effort estimates or schedules when using this model.

Code and Fix

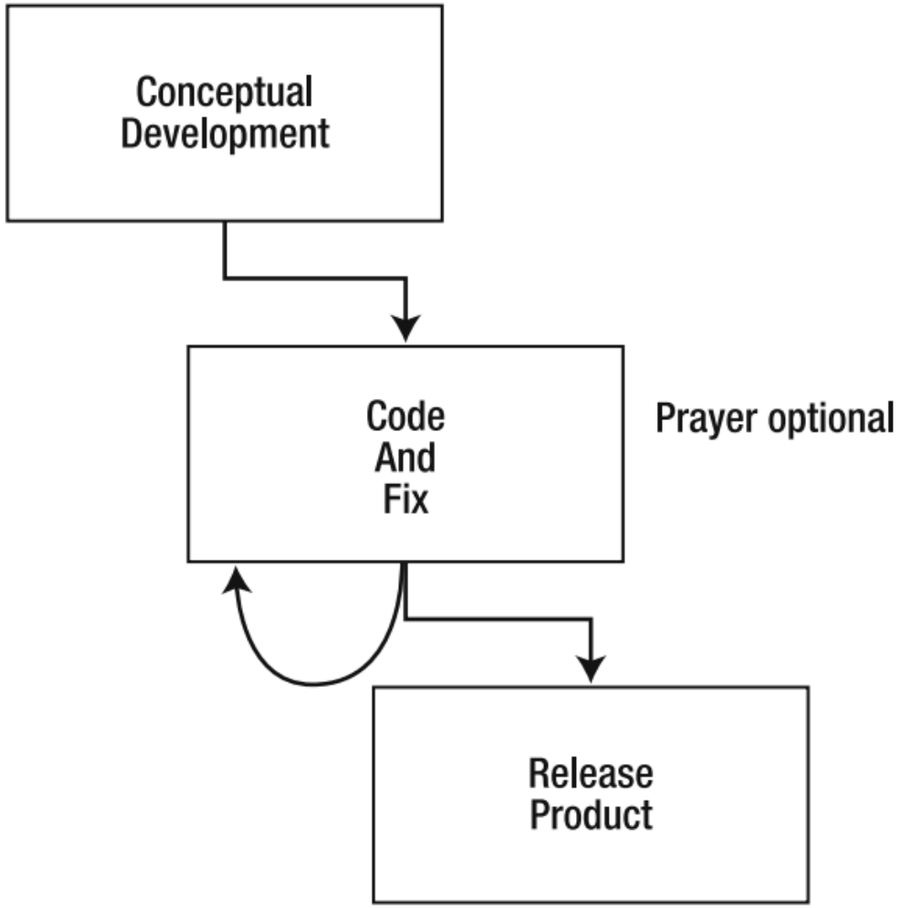


Figure 2-1. *The code and fix process (non) model*

Code and Fix (Cont.)

- Code and fix says take a minimal amount of time to understand the problem and then start coding.
- Compile your code and try it out. If it doesn’t work, fix the first problem you see and try it again.
- Continue this cycle of type-compile-run-fix until the program does what you want with no fatal errors and then ship it.
- It’s useful to validate architectural decisions and to show a quick version of a user interface design. Use it to understand the larger problem you’re working on.

The Waterfall

- The first and most traditional of the plan-driven process models is the **waterfall model**.
- It progresses nicely through requirements gathering and analysis, to architectural design, detailed design, coding, debugging, system testing, release, and maintenance.
- It requires **detailed** documentation at each stage, along with reviews, archiving of the documents, sign-offs at each process phase, configuration management, and close management of the entire project. It's a model of the plan-driven process.

The Waterfall

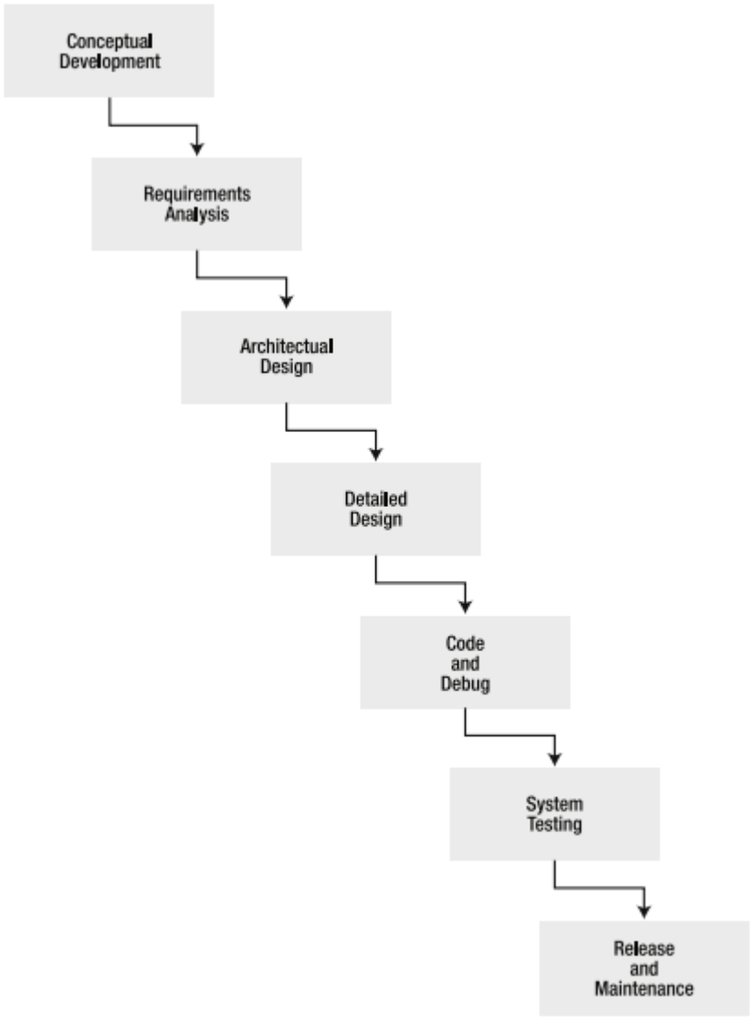


Figure 2-2. The waterfall process model

The Waterfall (Cont.)

- There are two fundamental problems with the waterfall model that hamper its acceptance and make it very difficult to implement:
  - First, it generally requires that you finish phase N before you continue on to phase N+1.
  - In the simplest example, this means that you must nail down all your requirements before you start your architectural design, and finish your coding and debugging before you start anything but unit testing, etc. In theory, this is great.
  - You'll have a complete set of requirements, you'll understand exactly what the customer wants, and everything the customer wants, so you can then confidently move on to designing the system.
- In practice, this never happens though.

The Waterfall (Cont.)

- The second problem with the waterfall is:
  - It has no provision for backing up. It is fundamentally based on an assembly-line mentality for developing software.
    - The nice little diagram shows no way to go back and rework your design if you find a problem during implementation.
    - The implications are that you really have to nail down one phase and review everything in detail before you move on.
- In practice this is just not – practical. The world doesn't work this way. You never know everything you need to know at exactly the time you need to know it.

The Waterfall (Cont.)

- Advantages:
  - It’s also a good way to start thinking about very large projects; it gives managers a warm fuzzy because it lets them think they know what’s going on (they don’t, but that’s another story).
  - It's also a good model for inexperienced teams working on a well-defined, new project because it leads them through the life cycle.

The waterfall with feedback

- A straight-line waterfall doesn’t work and that you need the ability to back up to a previous phase when you discover a problem in the current phase.
- **The waterfall with feedback** model recognizes that you have to start work with incomplete requirements, design, test plan, and so on.
- It also explicitly builds in the idea that you will have to go back to previous process steps as new information about your project is uncovered.
  - The new information can be **new** requirements, **updated** requirements, design flaws, defects in testing plans, and the like.
  - Any of these will require that you revisit a previous process step to rectify the problem.

The waterfall with feedback

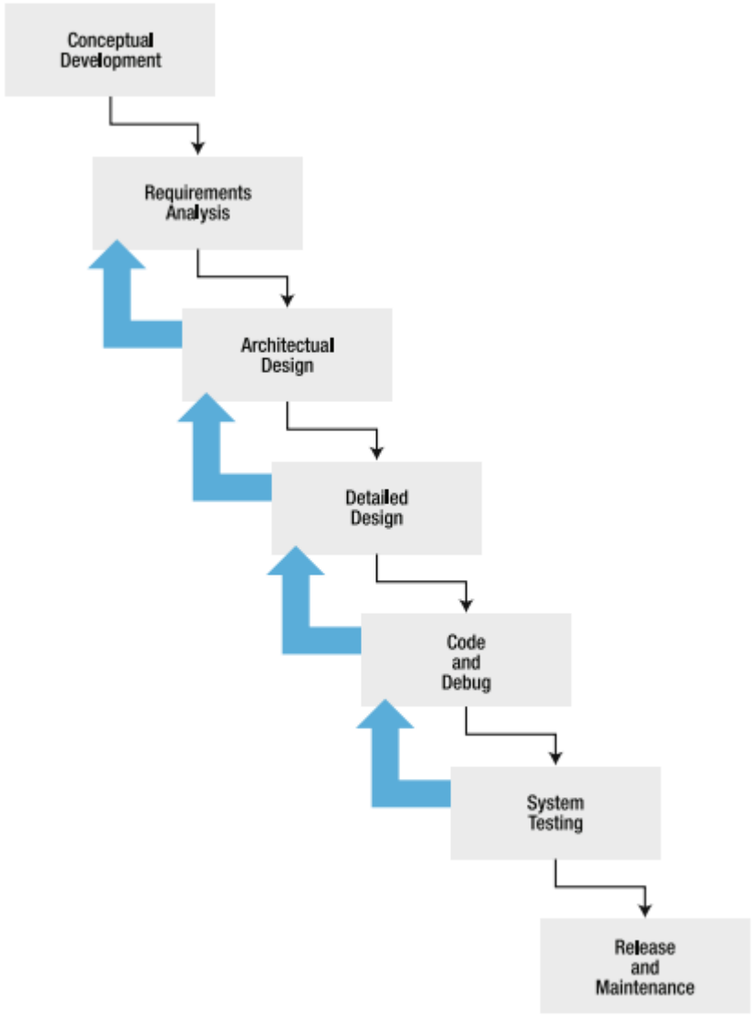


Figure 2-3. Waterfall with feedback process model

The waterfall with feedback (Cont.)

- This process model is still quite rigid, and it still has the same advantages of a waterfall model when it comes to very large, new projects and inexperienced teams.
- The two main disadvantages with the waterfall with feedback model are that it really messes with your scheduling big time, and it makes it harder to know when you're finished.
  - It messes with your schedule because in any phase there can be unexpected moves back to a previous phase of development.
  - This also means it's harder to know when you are done.
- Because of these disadvantages, the waterfall with feedback model also morphs into a new model, one that attempts to address the scheduling and uncertainty issues.

Loops Are Your Friend

The best practice is to iterate and deliver incrementally, treating each iteration as a closed-end “mini-project,” including complete requirements, design, coding integration, testing, and internal delivery. **On the iteration deadline**, deliver the (fully-tested, fully-integrated) system thus far to internal stakeholders.

Solicit their feedback on that work, and fold that feedback into the plan for the next iteration.

Iterative process models

- While the waterfall with feedback model recognizes that all the requirements aren't typically known in advance, and that mistakes will be made in architectural design and detailed design, it doesn't go far enough in taking those realizations into the process.
- Iterative process models make this required change in process steps more explicit and create process models that build products a piece at a time.
- In most iterative process models, you'll take the known requirements – you'll take a snapshot of the requirements at some time early in the process – and prioritize them, typically based on the customer's ranking of what features are most important to deliver first.

Iterative process models (Cont.)

- You then pick the highest priority requirements and plan a series of iterations, where each iteration is a complete project.
- **For each iteration:**
  - You'll add a set of the next highest priority requirements (including some you may have discovered during the previous iteration) and repeat the project.
- By doing a complete project with a subset of the requirements every time at the end of each iteration you end up with a **complete**, **working**, and **robust** product, albeit with fewer features than the final product will have.

Tom DeMarco basic rule

Your project, the whole project, has a binary deliverable. On the scheduled completion day, the project has either delivered a system that is accepted by the user, or it hasn't.

Everyone knows the result on that day.

The object of building a project model is to divide the project into component pieces, each of which has this same characteristic: each activity must be defined by a deliverable with objective completion criteria.

The deliverables are demonstrably done or not done."

Iterative process models (Cont.)

- So what happens if you estimate wrong?
- What if you decide to include too many new features in an iteration?
- What if there are unexpected delays?

Well, if it looks as if you won't make your iteration deadline there are only two realistic alternatives: **move the deadline, or remove features**. We'll come back to this problem when we talk about estimation and scheduling.

The key to iterative development is "live a balanced life" analyze some and design some and code some and test some every day.

Evolutionary prototyping process model

- The traditional way of implementing the incremental model is known as **evolutionary prototyping**.
- In **evolutionary prototyping**, one prioritizes requirements as they are received and produces a succession of increasingly feature-rich versions of the product.
- Each version is refined using customer feedback and the results of integration and system testing.
- This is an excellent model for an environment of changing or ambiguous requirements, or a poorly understood application domain.
- This is the model that evolved into **the modern agile development processes**.

Evolutionary prototyping process model (Cont.)

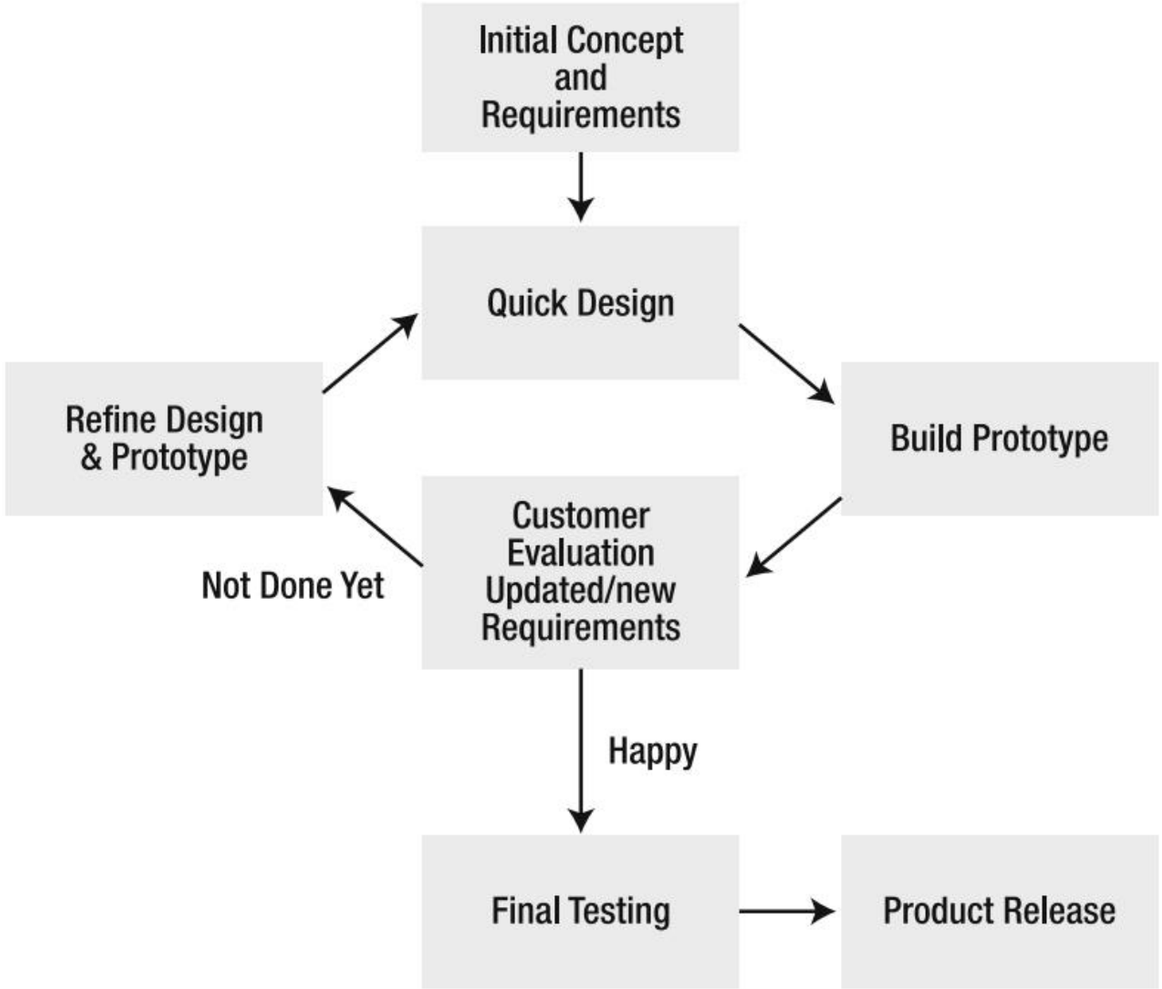


Figure 2-4. *Evolutionary prototyping process model*

Evolutionary prototyping process model (Cont.)

- Evolutionary prototyping provides improved progress visibility for both the customer and project management. It also provides good customer and end user input to product requirements and does a good job of prioritizing those requirements.
- **On the downside,** evolutionary prototyping leads to the danger of unrealistic schedules, budget overruns, and overly optimistic progress expectations.
  - These can happen because the limited number of requirements implemented in a prototype can give the impression of real progress for a small amount of work.
  - On the flip side, putting too many requirements in a prototype can result is schedule slippages, because of overly optimistic estimation.

Evolutionary prototyping process model (Cont.)

- There is also the possibility of low maintainability, again because the design and code evolve as requirements change. This may lead to lots of re-work, a busted schedule, and increased difficulty in fixing bugs post-release.
- Evolutionary prototyping works best with tight, **experienced teams** who have worked on **several projects together**.
  - This type of cohesive team is productive and dexterous, able to focus on each iteration and usually producing the coherent, extensible designs that a series of prototypes requires.
- This model is **not** generally recommended for **inexperienced teams.**

The agile model

- Starting in the mid 1990s, a group of process mavens began advocating a new model for software development.
- As opposed to the heavyweight plan-driven models mentioned above, this new process model was lightweight. It required **less documentation and fewer process controls**.
- It was targeted at small to medium-sized software projects and smaller teams of developers.
- It was intended to allow these teams of developers to quickly adjust to changing requirements and customer demands, and it proposed to release completed software much more quickly than the plan-driven models.
- It was, in a word, agile.

The agile model

- Agile development works from the proposition that the goal of any software development project is working code.
- And because the focus is on working software, then the development team should spend most of their time writing code, not writing documents. This gives these processes the name lightweight.
- Lightweight methodologies have several characteristics:
  - They tend to emphasize writing tests before code, frequent product releases, significant customer involvement in development, common code ownership, and refactoring – rewriting code to make it simpler and easier to maintain.
- Lightweight methodologies also suffer from several myths:
  - The two most pernicious are probably that lightweight processes are only good for very small projects, and that you don’t have to have any process discipline in a lightweight project.

The agile model

- The truth is that lightweight methodologies have been successfully used in many small and medium-sized projects – say up to about 500K lines of code.
- Lightweight methodologies also require process discipline, especially in the beginning of a project when initial requirements and an iteration cycle are created and in the test-driven-development used as the heart of the coding process.

We'll look at two lightweight/agile methodologies, eXtreme Programming, and Scrum.

eXtreme Programming (XP)

- eXtreme Programming was created around 1995 by Kent Beck and Ward Cunningham.
- XP is a “lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop software.”

XP Overview

- XP relies on the following four fundamental ideas:
  - **Heavy customer involvement:** XP requires that a customer representative be part of the development team and be on site at all times. The customer representative works with the team to create the contents of each iteration of the product, and she creates all the acceptance tests for each interim release.
  - **Continuous unit testing (also known as test-driven development [TDD]):** XP calls for developers to write the unit tests for any new features before any of the code is written. In this way the tests will, of course, initially all fail, but it gives a developer a clear metric for success. When all the unit tests pass, you've finished implementing the feature.

XP Overview

- XP relies on the following four fundamental ideas:
  - ***Pair programming*: XP requires that all code be written by pairs of developers.**
    - Pair programming requires two programmers – a driver and a navigator – who share a single computer. The driver is actually writing the code while the navigator watches, catching typos, making suggestions, thinking about design and testing, and so on.
    - The pair switches places periodically (every 30 minutes or so, or when one of them thinks he has a better way of implementing a piece of code).
    - Pair programming works on the “two heads are better than one” theory. While a pair of programmers is not quite as productive as two individual programmers when it comes to number of lines of code written per unit of time, their code usually contains fewer defects, and they have a set of unit tests to show that it works. This makes them more productive overall.
    - Pair programming also provides the team an opportunity to re-factor existing code – to re-design it to make it as simple as possible while still meeting the customer’s requirements. Pair programming is not exclusive to XP, but XP was the first discipline to use it exclusively.

XP Overview

- XP relies on the following four fundamental ideas:
  - **Short iteration cycles and frequent releases:** XP typically uses release cycles in the range of just a few months and each release is composed of several iterations, each on the order of 4–6 weeks.
    - The combination of frequent releases and an on-site customer representative allows the XP team to get immediate feedback on new features and to uncover design and requirements issues early.
    - XP also requires constant integration and building of the product. Whenever a programming pair finishes a feature and it passes all their unit tests, they immediately integrate and build the entire product. They then use all the unit tests as a regression test suite to make sure the new feature hasn't broken anything already checked in.
    - If it does break something, they fix it immediately. So in an XP project, integrations and builds can happen several times a day. This process gives the team a good feel for where they are in the release cycle every day and gives the customer a completed build on which to run the acceptance tests.

XP Motivation

- Risk is the most basic problem in software.
- Risk manifests itself in many ways: schedule slips, project cancelation, increased defect rates, misunderstanding of the business problem, false feature rich (you've added features the customer really doesn't want or need), and staff turnover.
- Managing risk is a very difficult and time-consuming management problem.
- Minimizing and handling risk are the key areas of risk management.

XP seeks to minimize risk by controlling the four variables of software development

XP (The Four Variables)

- The four variables of software development projects are as follows: Cost, Time, Features, and Quality.
- **Cost** is probably the most constrained; you can’t spend your way to quality or schedule, and as a developer you have very limited control over cost.
- **Time** is your delivery schedule and is unfortunately usually imposed on you from the outside.
- **Quality** is the number and severity of defects you are willing to release with. You can make shortterm gains in delivery schedules by sacrificing quality, but the cost is enormous.
- **Features** (also called scope) is what the product actually does. This is what developers should always focus on. It’s the most important of the variables from the customer’s perspective and it is also the one you as a developer have the most control over. Controlling scope allows you to provide managers and customers control over quality, time, and cost.

XP (The Four Variables)

- XP recognizes that to minimize risk, developers need to control as many of the variables as possible, but especially they need to control the scope of the project.
- XP uses the metaphor of “learning to drive.” Learning to drive is not pointing the car in the right direction. It’s pointing the car, constantly paying attention and making the constant minor corrections necessary to keep the car on the road.
- In programming, the only constant is change. If you pay attention and cope with change as it occurs, you can keep the cost of change manageable.

XP (The Four Values)

- In order for XP to be a viable discipline of development everyone who is involved in an XP project needs to buy into a common set of values that will permeate all the rules that make up the discipline.
- In XP there are the following four core values that enable it to work:
  - Communication
  - Simplicity
  - Feedback
  - Courage

XP (The Four Values)

- **Communication** really means spreading the collective knowledge of the group around to all the members. Keeping the XP team small facilitates communication by keeping the number of lines of communication small.
  - Pair programming and collective ownership of the code also facilitate communication by spreading the knowledge of the entire code base around the entire team.
- **Simplicity is key.** XP focuses on developing the simplest piece of software that solves today's task.
  - XP developers bet that "...it is better to do a simple thing today and pay a little more tomorrow to change it if it needs it, than to do a more complicated thing today that may never be used anyway."

XP (The Four Values)

- All developers on an XP team are allowed and encouraged to redesign code to make it simpler at any time. This practice is called “**refactoring**.”

- XP programmers are required to write tests before they write the code, so that they always have immediate feedback about their code and its impact on the system.

- XP developers must have courage. They must be willing to make changes at any time when the design no longer fits. They need to be prepared to throw code away if it doesn’t work.

- Simplicity supports courage because you’re less likely to break a simple system.

- XP team members track the schedule daily and involve the customer in re-prioritizing features as soon as needed.

XP(The 15 Principles)

- From the four values described above XP derives some basic principles. The list looks like the following:
  - **Rapid feedback:** Get feedback, interpret it, and put it back into the system as quickly as possible.
  - **Assume simplicity:** Focus on today's task and solve it in the simplest way possible.
  - **Incremental change:** Integrate your new code into the system every day.
  - **Embracing change:** It's gonna happen, so be prepared for it.
  - **Quality work:** Quality isn't free; strive for defect-free code. Pair programming gives you the two heads are better than one gift and test-driven development focuses your code on satisfying requirements. Both of these help lead to fewer defects in your code.

XP(The 15 Principles)

- From the four values described above XP derives some basic principles. The list looks like the following:
  - **Teach learning:** Teach how to learn to do testing, refactoring, and coding better rather than set down a set of rules that say, “you must test this way.”
  - **Small initial investment:** The emphasis here is on small teams, particularly at the beginning of a project to manage the resources carefully and conservatively.
  - **Play to win:** As opposed to playing not to lose. If you don’t worry about schedules, or requirements churn, your days will be more relaxed, you’ll be able to focus on the problems at hand (and not on the next deadline) and your code will be cleaner, you’ll be more relaxed and more productive. Just relax and win.
  - **Concrete experiments:** Every abstract decision (requirements or design) should be tested.
  - **Open, honest communication:** You have to be able to criticize constructively and be able to deliver bad news as well as good.
  - **Work with people’s instincts, not against them:** People generally like to win, like working with others, like being part of a team, and especially like seeing their code work. Don’t do things that go against this.

XP(The 15 Principles)

- From the four values described above XP derives some basic principles. The list looks like the following:
  - **Accepted responsibility:** The team as a whole is responsible for the product. XP teams typically do not have managers that assign work; they have a coach to help with the process and a project manager to take care of the administrative tasks.
  - **Local adaptation:** Change XP to fit your local circumstances and project.
  - **Travel light:** The team and process artifacts you maintain should be few, simple, and valuable.
  - **Honest measurement:** Measure at the right level of detail and only measure what makes sense for your project.

Software Development and Professional Practice

Lecture 3: Process Life Cycle Models (Part 2)

For
Faculty of Computers and Information
Assiut University

By
Dr. Mostafa Salem
Assistant Professor, FCI-Assiut University
https://mostafasalem.netlify.app

The Four Basic Activities

- In order for XP to take the values and principles just described and create a discipline out of them, we need to describe the activities we'll use as the foundation.
- XP describes four activities that are the bedrock of the discipline.
  - Coding.
  - Testing.
  - Listening
  - Designing.

The Four Basic Activities

- **Coding:** The code is where the knowledge of the system resides so it's your main activity.
  - The fundamental difference between plan-driven models and agile models is this emphasis on the code.
  - In a plan-driven model, the emphasis is on producing a set of work products that together represent the entire work of the project with code being just one of the work products.
  - In agile methodologies, the code is the sole deliverable and so the emphasis is placed squarely there; in addition, by structuring the code properly and keeping comments up to date, the code becomes documentation for the project.

The Four Basic Activities

- **Testing:** The tests tell you when you are done coding.
  - Test-driven development is crucial to the idea of managing change.
  - XP depends heavily on writing unit tests before writing the code that they test and on using an automated testing framework to run all the unit tests whenever changes are integrated.

The Four Basic Activities

- **Listening:** To your partner and to the customer.
  - In any given software development project there are two types of knowledge:
    - The customer has knowledge of the business application being written and what it is supposed to do. **This is the domain knowledge of the project.**
    - The developers have knowledge about the target platform, the programming language(s), and the implementation issues. **This is the technical knowledge of the project.**
    - The customer doesn't know the technical side and the developers don't have the domain knowledge, so listening – on both sides – is a key activity in developing the product.

The Four Basic Activities

- **Designing:** Design while you code. “Designing is creating a structure that organizes the logic in the system.
  - **Good** design organizes the logic so that a change in one part of the system doesn’t always require a change in another part of the system.
  - **Good** design ensures that every piece of logic in the system has one and only one home.
  - **Good** design puts the logic near the data it operates on.
  - **Good** design allows the extension of the system with changes in only one place.

Implementing XP: The 12 Practices

- We (finally) get to the implementation of XP. Here are the rules that every XP team follows during their project.
- The rules may vary depending on the team and the project, but in order to call yourselves an XP team, you need to do some form of these things.
- The practices described here draw on everything previously described: **the four values**, **the 15 principles**, **and the four activities**.
- This is really XP

Implementing XP: The 12 Practices

- **The planning game:** Develop the scope of the next release by combining business priorities and technical estimates.
- The customer and the development team need to decide on the stories (read features) that will be included in the next release, the priority of each story, and when the release needs to be done.
- The developers are responsible for breaking the stories up into a set of tasks and for estimating the duration of each task.
- The sum of the durations tells the team what they really think they can get done before the release delivery date. **If necessary, stories are moved out of a release if the numbers don't add up.**
- Notice that estimation is the responsibility of the developers and not the customer or the manager.
- In XP only the developers do estimation.

Implementing XP: The 12 Practices

- **Small releases:** Put a simple system into production quickly, and then release new versions on a very short cycle.
  - Each release has to make sense from a business perspective, so release size will vary. It is far better to plan releases in durations of **a month or two rather than six or twelve**.
  - The longer a release is, the harder it is to estimate.
- **Metaphor:** “A simple shared story of how the whole system works.”
  - The metaphor replaces your architecture. It needs to be a coherent explanation of the system that is decomposable into smaller bits – stories.
  - Stories should always be expressed in the vocabulary of the metaphor and the language of the metaphor should be common to both the customer and the developers.

Implementing XP: The 12 Practices

- **Simple design:** Keep the design as simple as you can each day.
  - Re-design often to keep it simple.
  - A simple design:
    - (1) runs all the unit tests,
    - (2) has no duplicated code,
    - (3) expresses what each story means in the code, and
    - (4) has the fewest number of classes and methods that make sense to implement the stories so far.

Implementing XP: The 12 Practices

- **Testing:** Programmers constantly write unit tests.
  - Tests must all pass before integration.
  - Although this works for most acceptance tests and should certainly work for all unit tests, this analogy breaks down in some instances, notably in testing the user interface in a GUI.
  - Even this can be made to work automatically if your test framework can handle the events generated by a GUI interaction.

Implementing XP: The 12 Practices

- **Refactoring:** Restructure the system “without changing its behavior” to make it simpler – remove redundancy, eliminate unnecessary layers of code, or to add flexibility.
  - The key to refactoring is to identify areas of code that can be made simpler and to do it while you’re there.
  - Refactoring is closely related to collective ownership and simple design. Collective ownership gives you permission to change the code and simple design imposes on you the responsibility to make the change when you see it needs to be made.

Implementing XP: The 12 Practices

- **Pair programming:** All production code written in an XP project must be written by two programmers at one machine. Any code written alone is thrown away.
  - Pair programming is a dynamic process. You may change partners as often as you change tasks to implement. This has the effect of reinforcing collective ownership by spreading the knowledge of the entire system around the entire team.
  - It avoids the “beer truck problem,” where the person who knows everything gets hit by a beer truck and thus sets the project schedule back months.

Implementing XP: The 12 Practices

- **Collective ownership:** The team owns everything, implying that anyone can change anything at any time. In some places this is known as “ego-less programming.”
    - Programmers need to buy into the idea that anyone can change their code and that collective ownership extends from code to the entire project; it’s a team project, not an individual one.
- **Continuous integration:** Integrate and build every time a task is finished, possibly several times a day (as long as the tests all pass).
    - This helps to isolate problems in the code base; if you’re integrating a single task change, then the most likely place to look for a problem is right there.

Implementing XP: The 12 Practices

- **40-hour week:** Work a regular 40-hour week. Never work a second week in a row with overtime. The XP philosophy has a lot in common with many of Tom DeMarco's Peopleware arguments.
  - People are less productive if they're working 60 or 70 hours a week than if they are working 40 hours.
  - When you're working excessive amounts of overtime, several things happen. Because you don't have time to do chores and things related to your "life," you do them during the **workday**.
  - Constantly being under deadline pressure and never getting a sustained break also means you get tired and then make more mistakes, which somebody then needs to fix.
  - But being in control of the project and working 40-hours a week (give or take a few) **leaves you with time for a life**, **time to relax and recharge**, and **time to focus on your work during the work-day**, making you more productive, not less.

Implementing XP: The 12 Practices

- **On-site customer:** A customer is part of the team, is on-site, writes and executes functional tests, and helps clarify requirements.
  - The customer’s ability to give immediate feedback to changes in the system also increases team confidence that they are building the right system every day.
- **Coding standards:** The team has ‘em, follows ‘em, and uses 'em to improve communication.
  - Because of collective code ownership the team must have coding standards and everyone must adhere to them.
  - Without a sensible set of coding guidelines, it would take much, much longer to do refactoring and it would decrease the desire of developers to change code.
  - Your coding standards should make your code easier to read and maintain: they shouldn’t constrict creativity.

The XP Life Cycle

The XP life cycle contains all the phases of the generic life cycle described at the start of the chapter

(1- Conception. 2- Requirements gathering/exploration/modeling. 3- Design. 4- Coding and debugging.

5- Testing. 6- Release. 7- Maintenance/software evolution. 8- Retirement), but

- It compresses the middle three **phases – design, code, and test – into a single implementation phase**.
- A productizing phase is added after implementation to allow the code to be **stabilized** before release.
- Then 6 more phases were added

The XP Life Cycle

- ***9- Exploration***:
  - Exploration is done when “the customer is confident that there is more than enough material on the story cards to make a good first release and the programmers are confident that they can’t estimate any better without actually implementing the system.”
  - During exploration, the team’s main goal is to get as many requirements (story cards) written as they can.
  - This is also the time when they can explore the architecture possibilities by doing a quick spike of the system. Estimate all tasks done during exploration to practice your estimation skills. In most projects Exploration is the “fuzzy front-end” of the project. You’re not quite sure how long it will take and you’re gathering requirements and trying to figure out what the product will actually *do*.

The XP Life Cycle

- **10- Planning game:** The Planning game is the tail end of your release exploration phase.
  - In the planning game you need to identify your top priority, high-value stories and agree with the customer which ones will be in the next release.
  - Releases should be from two to six months duration each. Any shorter and you're not likely to get any significant work done and any longer is just plain too hard to plan. Then you need to plan the first few iterations for the release; iterations are 1 to 4 weeks each.
  - Each iteration produces functional test cases for each story scheduled for the iteration. The first iteration helps you nail down your metaphor for the project and puts the architecture in place.
  - Subsequent iterations add new features based on the prioritized list of stories. Reschedule as necessary.

The XP Life Cycle

- **11- Implement:** Design, code, test, or actually, design, test, code. One task at a time until all the tasks for a story are complete, and one story at a time until all the stories for this iteration are complete.

- **12- Productizing:** Occurs in the last iteration before your release is done. At this point you should freeze new functionality and focus on stabilizing the product, tuning performance, if necessary, and running acceptance tests.

- **13- Maintenance/evolution:** Well, according to the agile philosophy, you're always in maintenance mode. Here though, you've released something the customer will use and you now must "simultaneously produce new functionality, keep the existing system running, incorporate new people into the team, and bid farewell to members who move on."

- **14- Death:** If the customer can't come up with new stories, mothball the code. If the system can't deliver anymore, mothball the code and start over.

Scrum

- The second agile methodology we'll look at is **Scrum.**
- **Scrum** derives its name from rugby, where a scrum is a means of restarting play after a rules infraction.
- The scrum uses the eight forwards on a rugby team (out of 15 players in the rugby union form of the game) to attempt to (re)gain control of the ball and move it forward towards the opposing goal line.
- **The idea in the agile Scrum methodology** is that a **small** team is unified around a single goal and gets together for sprints of development that move them towards that goal.
- **Scrum** is a variation on the iterative development approach and incorporates many of XP features.
- **Scrum** is **more of a management approach** than XP and **doesn't define** many of the detailed development practices (like pair programming or test-driven development) that XP does, although most scrum projects will use these practices.

Scrum

- **Scrum** uses teams of no more than 10 developers. Just like other agile methodologies, scrum emphasizes the efficacy of small teams and collective ownership.

- **Scrum** is characterized by the sprint, an iteration of between one and four weeks. Sprints are timeboxed in that they are of a fixed duration and the output of a sprint is what work the team can accomplish during the sprint.

- The delivery date for the sprint does not move out. This means that sometimes a sprint can finish early, and sometimes a sprint will finish with less functionality than was proposed. A sprint always delivers a usable product.

Scrum

- **Scrum requirements** are encapsulated in two backlogs.
  - The product backlog is the prioritized list of all the requirements for the project; it is created by the scrum team and the product owner.
  - The sprint backlog is the prioritized list of requirements (say user stories) for the current sprint.
  - Once the sprint starts, only the development team may add items to the sprint backlog – these are usually **bugs** found during testing. **No** outside entity may add items to the sprint backlog, only to the product backlog.
- Scrum projects are facilitated by **a ScrumMaster** whose job it is to manage the backlogs, run the daily Scrum meetings, and to protect the team from outside influences during the sprint. **The scrum master is usually not a developer.**

Scrum

- **Scrum projects** have a daily scrum meeting, which is a stand-up meeting of 15–30 minutes duration where the entire team discusses sprint progress.
- **The daily Scrum meeting** allows the team to share information and track sprint progress. By having daily Scrum meetings, any slip in the schedule or any problems in implementation are immediately obvious and can then be addressed by the team at once.
- **The Scrum master** ensures that everyone makes progress, records the decisions made at the meeting and tracks action items, and keeps the Scrum meetings short and focused.
- At the Scrum meeting, each team member answers the following three questions in turn:

1. What tasks have you finished since the last Scrum meeting?

2. Is anything getting in the way of your finishing your tasks?

3. What tasks are you planning to do between now and the next Scrum meeting?

Scrum

- Discussions other than responses to these three questions are deferred to other meetings.
  - This meeting type has several effects.
    - It allows the entire team to visualize progress towards the sprint and project completion every day.
    - It reinforces team spirit by sharing progress – everyone can feel good about tasks completed.
    - It verbalizes problems – which can then be solved by the entire team.
- The development team itself is self-organizing;
  - The members of the Scrum team decide among themselves who will work on what user stories and tasks, assume collective ownership of the project, and decide on the development process they'll use during the sprint.
  - This organization is reinforced every day at the Scrum meeting.

Scrum (Sprint)

- **Before the first sprint starts,**
  - Scrum has an **initial** planning phase that creates the list of the initial requirements, decides on an architecture for implementing the requirements, divides the user stories into prioritized groups for the sprints, and breaks the first set of user stories into tasks to be estimated and assigned.
  - They stop when their estimates occupy all the time allowed for the sprint.
  - Tasks in a sprint should not be longer than one day of effort.
- **After each sprint**, another planning meeting is held where the **Scrum master** and the **team** reprioritize the product backlog and create a backlog for the next sprint.
- With most Scrum teams, estimates of tasks become better as the project progresses primarily because the team now has data on how they have done estimating on previous sprints.
- This effect in Scrum is called "**acceleration**;" the productivity of the team can actually increase during the project as they gel as a team and get better at estimating tasks. This planning meeting is also where the organization can decide whether the project is finished, or whether to finish the project at all.

Scrum (Sprint)

- **After the last scheduled sprint,** a final sprint is done to bring closure to the project. This sprint implements no new functionality, but prepares the final deliverable for product release.
  - It fixes any existing bugs, finishes documentation, and generally productizes the code.
  - Any requirements left in the product backlog are transferred to the next release.
  - A Scrum retrospective is held before the next sprint begins to ponder the previous sprint and see if there are any process improvements that can be made.
  - Scrum is a project management methodology and is typically silent on development processes.
  - Despite this, Scrum teams typically use many of the practices described above in the XP practices section.
  - Common code ownership, pair programming, small releases, simple design, test-driven development, continuous integration and coding standards are all common practices in Scrum projects.

Software Development and Professional Practice

Lecture 4: Project Management Essentials

For
Faculty of Computers and Information
Assiut University

By
Dr. Mostafa Salem
Assistant Professor, FCI-Assiut University
https://mostafasalem.netlify.app

Introduction

- Learning something about project management from both sides is an essential part of learning software development.
- Project management is an involved and complicated set of tasks. We'll restrict ourselves to several tasks that will impact you as a developer the most. They are the following:
  - Project planning
  - Estimation and scheduling
  - Resource management
  - Project oversight
  - Project reviews and presentations

Project Planning

- Project planning is forever. By that I mean that project planning continues throughout the entire duration of the project.
- **The plan:**
  - In most projects, and especially in those that are using a plan-driven process model, a project plan is an actual document that is written by the project manager, and that is approved and signed off on by the development team and by upper management.
  - It is, in effect, a contract, of what the team is going to do and how they are going to do it.
  - It also says how the project will be managed, and in the extreme plan-driven cases, even states how and when the document itself will be modified.

The plan

- What’s in the project plan? A project plan consists of the following seven parts:
  - Introduction and explanation of the project
  - Project organization
  - Risk analysis
  - Hardware, software, and human resource requirements
  - Work breakdown and task estimates
  - Project schedule
  - Project monitoring and reporting mechanisms, collectively known as project oversight
- Not all of these are necessary for all projects or project methodologies. In particular, plan-driven projects will use all of them, while agile projects may use a few on a single page.

The plan

Project plans are a great tool for setting down what you think you're doing, an outline of how it will be done, and how you plan on executing the outline.

The problem with a project plan is that it's static.

Once it's written and signed off on, upper management thinks the project will run exactly as stated in the plan. But the reality of the project often thwarts the plan.

Project Organization

- The project organization section of the plan contains the following three things:
  - How you're going to organize the team?
  - What process model the project will be using ?
  - How will the project be run on a day-to-day basis ?
- If you're working with an experienced team, all this is already known to everyone, so your project organization section can be, "We'll do what we usually do."
- However, this section is a necessity for inexperienced teams.

Risk Analysis

- In the risk analysis section, you need to think about the bad things.
  - What can possibly go wrong with this project?
  - What is the worst that could happen?
  - What will we do if it does?

Risk Analysis

- Some risks to watch out for are:
  - **Schedule slips:** That task that you estimated would take three days has just taken three weeks.
    - **In a plan-driven project,** this can be an issue if you don't have regular status meetings. Waiting three weeks to tell your boss that you're late is always worse than telling her that you'll be late as soon as you know it.
    - **In an agile project** this is unlikely, because most agile projects have a daily status meeting.

Risk Analysis

- Some risks to watch out for are:
  - **Defect rate is excessive:** Your testing is finding lots of bugs.
    - What do you do, continue to add new features or stop to fix the bugs?
    - This can be a real issue in a project where integration builds happen according to a fixed schedule, say once a week.
    - In a project where integrations happen every day, you can keep up with defects more easily.
    - In either case, if you are experiencing a high defect rate, the best thing to do is to stop, take a look around, and find the root cause of the defects before adding more functionality.

Risk Analysis

- Some risks to watch out for are:
  - **Requirements misunderstood:** What you're doing isn't what the customer wanted.
    - This classic problem is the result of the fact that customers and developers live in two different worlds.
    - The customer lives in the application domain.
    - The developer understands from a technical perspective how the product will work.
    - The best way to avoid this situation is to have the customer on site as often as possible and to produce deliverable products as often as possible.

Risk Analysis

- Some risks to watch out for are:
  - **Requirements churn:** New features, altered features, deleted features … will the misery never end?
    - **Requirements churn** is probably the largest single reason for missed delivery dates, high defect rates, and project failure.
    - Churn happens when the customer (or the development team itself) continues to change requirements while development is underway.
    - It leads to massive amounts of rework in the code, retesting of baselines, and delay after delay.
    - In a plan-driven process this is usually accomplished by a change control board (CCB) that examines each new requirement and decides whether to add it to the list of features to be implemented.
    - There may be a member of the development team on the CCB, but that's not required, so the danger here is that the CCB will add new features without understanding all the scheduling and effort ramifications.
    - In agile processes, the development team usually keeps control of the prioritized requirements list (called the product backlog in Scrum), and only adjusts the list at set points in the project – after iterations in XP, and after each sprint in Scrum.

Risk Analysis

- Some risks to watch out for are:
  - **Turnover:** (the rate at which employees leave a workforce and are replaced)
    - Your most experienced developer decides to join a start-up three weeks before product delivery.
    - The best way to reduce turnover is to (1) give your developers interesting work, (2) have them work in a pleasant environment, and (3) give them control over their own schedules.
    - Money is not one of the main motivators for software developers. This doesn't mean they don't want to get paid well, but it does mean that throwing more money at them in order to get them to work harder or to keep them from leaving doesn't generally work.
    - The best way to mitigate the effect of turnover is to spread the knowledge of the project around all the members of the development team.
    - Principles like common code ownership and techniques like pair programming work to invest all the team members in the product and spreads the knowledge of the code across the entire team.

Risk Analysis

- Once you've got a list of the risks to your project, you need to address each one and talk about two things: avoidance and mitigation.
  - For each risk, think about how you can avoid it. Build slack into your schedule, do constant code reviews, freeze requirements early, do frequent releases, require pair programming so you spread around the knowledge of the code, and the like.
- Then you need to think about what you'll do if the worst-case scenario does happen; this is mitigation.
  - Remove features from a release, stop work on new features and do a bug hunt, negotiate new features into a future release, and so on.

Risk Analysis

- Once you address avoidance and mitigation, you'll have a plan on how to handle your identifiable risks.
- This doesn't completely let you off the hook, because there are bound to be risks you miss; but the experience of addressing the risks you do think of will enable you to better handle new ones that surprise you during the project.
- If your project is using an iterative process model, it's a good idea to revisit your risks after every iteration and see which ones have changes, identify any new ones, and remove any that can no longer happen.

Resource Requirements

- How many people do you need for the project?
- Do they all need to start at once, or can their starting dates on the project be staggered as phases are initiated?
- How many computers do you need? What software will you be using for development? What development environment do you need?
- Is everyone trained in that environment? What support software and hardware do you need?
- Many of these resource questions are usually answered for you by the platform you're targeting and the application domain in which you are working.
- Questions about team size, start dates, and phases of the project will likely not be able to be answered until you do a first cut at **effort estimation and scheduling.**

Work Breakdown and Task Estimates

Don’t ever believe anyone who tells you, “that feature will take six months to do.” That is a wildassed guess (WAG), and bears little to no relation to reality.

You just can’t estimate something that big. The best you can do is say, “I once implemented a feature like that in six months.”

And even that only helps a little.

Work Breakdown and Task Estimates

- Get your work broken down into tasks that are no more than about a week in duration.
- Never do estimation in any unit except person-hours.
- Once you have a believable list of tasks, you can start doing size and then effort estimation.
- Size always needs to come first, because you just can't figure out how long something will take until you have an idea of how big it is.
- Size can be several things, depending on your work breakdown and your development model;
  - Functional modules, number of classes, number of methods, number of function points, number of object points, or that old standby, uncommented lines of code.
  - Actually, no matter what you initially measure size in, you'll end up with estimates in terms of **KLOC** – thousands of uncommented lines of code.

Work Breakdown and Task Estimates

- **The Delphi method** is a quick and relatively efficient estimation technique. Here's one way it can work:
  - Find three of your most senior developers – these are the folks who've got the most experience, and who should therefore be able to give you a good guess.
  - Then give them the task breakdown.
  - Then ask them to give you three numbers for each task, the shortest amount of time it should take, the longest amount of time it should take, and the "normal" amount of time it should take, all in person-hours.
  - Once you have these numbers, add them all up, the shortest together, the longest together, and the "normal" together and take the mean. Those are your estimates for each task.
  - The averages of the best guess by your best developers for each task.
  - Pick one of the three values for each task as the official (for now) effort estimate and proceed to create a schedule.

Project Schedule

- Once you have estimates of the tasks in your first release or iteration and have people resource estimates, you can create a schedule.
- There are several things to take into account :Get your developers to tell you the dependencies between tasks.
  - Figure out what your duty cycle is.
  - Take weekends, vacations, sick days, training, and slack into account when you're making the schedule.
  - You can't schedule a developer to work on two tasks at the same time.
- Finally, use project-scheduling software to make your schedule like Microsoft Project, Fast Track Scheduling, or Merlin provide lots of features that make keeping the schedule up to date much easier.

Project Oversight

- Project oversight is what happens once you've got a schedule.
- Once your project begins, the work needs to be managed. How this happens depends on the process you're using. But regardless of the process you need to manage the schedule, manage the developers, manage the process itself, and above all, manage your manager.
  - Fear is not a motivator.
  - If your people aren't happy, you don't have a hope.
  - Treat your developers as humans, not resources.
  - Supporting your team and keeping them insulated from distractions is your number one job.
  - Remember, projects are cooperative, social events.

Status Reviews and Presentations

- Status reviews and presentations are an inescapable part of any project.
- The bigger the project, the more formal the review.
- Remember that reporting status doesn't fix problems, and that generally upper management doesn't like hearing about problems.
- When you give a project status report just tell where your project is and where it's going during the period before the next status report.
- Don't make excuses; be honest about problems and where you are in the schedule.
- Just providing good news is usually bad for your reputation; something will go wrong at some point, so it is best to get it out of the way right away.
- You must communicate bad news about the project as soon as possible. That's the best way to mitigate the problem and get others involved in helping to find a solution.

Status Reviews and Presentations

- When giving a presentation, be it a status review or a technical presentation, make sure you know your audience. Set your presentation to the level of the audience and keep the purpose of your presentation in front of you and them at all times.
  - **PowerPoint is ubiquitous in industry so learn to use it effectively.**
  - Keep your PowerPoint presentations short and to the point.
  - Avoid cramming your slides with lots of bullet points. Do not make your bullet points complete sentences, mostly because you'll be tempted to read them.
  - This is the kiss of death for two reasons: it takes too long and takes attention away from what you're actually saying.
  - Your bullet points should be talking points that you can then expand upon. This lets your audience focus on you, the speaker, rather than the slides. When you're constructing a PowerPoint presentation, use as few words as you can.

Defects

- Inevitably, you'll introduce defects (errors) into your program.
- As a developer, your aim is twofold:
  - Introduce as few defects as possible into the code you write.
  - Find as many of them as you can before releasing the code.

Defects

- Most development organizations have a set of defect levels they use to characterize just how bad a defect really is. One set of levels looks like the following:
  - **Fatal:** Either this defect causes the product to crash, or a fundamental piece of functionality doesn't work.
  - **Severe:** A major piece of functionality doesn't work, and there is no workaround for it that the user can perform.
  - **Serious:** A piece of functionality doesn't work, but there is a workaround for it that the customer can perform.
  - **Annoying:** A minor defect or error in the documentation that may annoy the user, but doesn't affect how the program works.
  - **New Feature Request:** This isn't a defect, but a request for the product to do something new.
- In nearly all organizations, no product can release with known level 1 or level 2 defects in it. Most organizations also try their best to remove all the level 3 defects as well.

Software Development and Professional Practice

Lecture 4: Software Architecture

For

Faculty of Computers and Information

Assiut University

By

Dr. Mostafa Salem

Assistant Professor, FCI-Assiut University

https://mostafasalem.netlify.app

Introduction

- What do we mean by a software **architecture**?
  - The term architecture conveys a notion of the core elements of the system, the pieces that are difficult to change.
  - A foundation on which the rest must be built.

Introduction

- There are really **two** levels of software design.
  - The level we normally think of when we're writing programs is usually called **detailed** design.
    - What operations do we need? What data structures? What algorithms are we using? How is the database going to be organized? What does the user interface look like? What are the calling sequences?
  - These are all very detailed questions that need to be answered **before** you can really get on with the detailed work of coding.

Introduction

- But there's **another** level of design. This kind of design is all about **style**.
  - If you were building a house, this design level asks questions like ranch or multi-story? Tudor or Cape Cod? Which direction do the bedroom windows face? Forced-air or hot-water heat? Three bedrooms or four? Open concept floor plan or closed?
  - These questions focus somewhat on details, but they are much more about the style of the house and how you'll be using it, rather than things like 12 or 14 gauge wire for the electrical system or the diameter of the air conditioning ductwork.
- This emphasis on **style** is what software architecture is all about.
- Software architecture is a set of **ideas** that tells you which **foundation** is the right one for your program.

Introduction

- The idea of software architecture began as a response to the increasing **size** and **complexity** of programs.
  - “As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation: designing and specifying the overall system structure emerges as a new kind of problem....
  - **This is the software architecture level of design.**

Introduction

- However, it is really the case that all programs of any size and complexity have an architecture.
- It’s just that for **larger** programs you need to be more intentional about your thinking about the architecture to make sure you have the right set of architectural patterns incorporated in your system design.

Introduction

- However, it is really the case that all programs of any size and complexity have an architecture.
- It’s just that for larger programs you need to be more intentional about your thinking about the architecture to make sure you have the right set of architectural patterns incorporated in your system design.
- The **architectural style** used for a program depends on what it is you’re **doing**.
- Different types of programs in different domains will lead us to different architectural styles; we can also call these architectural patterns since they have many characteristics of the design patterns.

General Architectural Patterns

- Whenever a software architect starts thinking about an architecture for a program, he usually starts by drawing pictures.
  - **Diagrams** of the architecture allow people to see the structure and framework of the program much more easily than text.
  - Software architectures are normally represented as **black box** graphs where graph nodes are computational structures, and the graph edges are communication **conduits** between the structures.

Introduction

- The conduits can represent data flow, object message passing, or procedure calls.
  - The United Modeling Language (UML).
  - Visual descriptions of architectures are generally easier to understand.

Pipe-and-filter Architecture

- In a **pipe-and-filter** style architecture, the computational components are called filters and they act as transducers that take input, transform it according to one or more algorithms, and then output the result to a communications conduit.

Figure 5-1. *The pipe-and-filter architecture*

Pipe-and-filter Architecture

Figure 5-1. The pipe-and-filter architecture

- The filters must be independent components.
- The classic example of a pipe-and-filter architectural style is **the Unix shell**, where there are a large number of small programs that typically do a single thing and can be **chained** together using the Unix **pipe** mechanism.
- Ex:
  - *The Problem*: Given a dictionary of words in English, find all the anagrams in the dictionary. That is, find all the words that are permutations of each other. For example, “pots,” “stop,” and “spot” are anagrams of each other.

Pipe-and-filter Architecture

1. Create a sign for each word in the list by sorting the letters in each word; keep the sign and the word together.

2. Sort the resulting list by the signs; all the anagrams should now be together.

3. Squash the list by putting each set of anagrams on the same line, removing the signs as you do.

```
sign <dictionary.txt | sort | squash >anagrams.txt
```

Pipe-and-filter Architecture

- Note that this example has all the features of a standard pipe-and-filter architecture:
  - **Independent** computational components that perform a transformation on their input data and,
  - Communication conduits that transmit the data from the output of one component to the input of the next.
- Note also that **not all applications** should use the pipe-and-filter architecture.
  - For example, it won't work so well for **interactive** applications or applications that respond to events or interrupts.

An Object-Oriented Architectural Pattern

- The advent of **object-oriented** analysis, design, and programming in the early 1980s brought with it several architectural and design patterns.
- **The Model-View-Controller (MVC)** architectural pattern is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller.
- MVC was originally developed to map the traditional input, processing, output roles of many programs into the GUI realm:

Input ➤ Processing ➤ Output

Controller ➤ Model ➤ View

An Object-Oriented Architectural Pattern

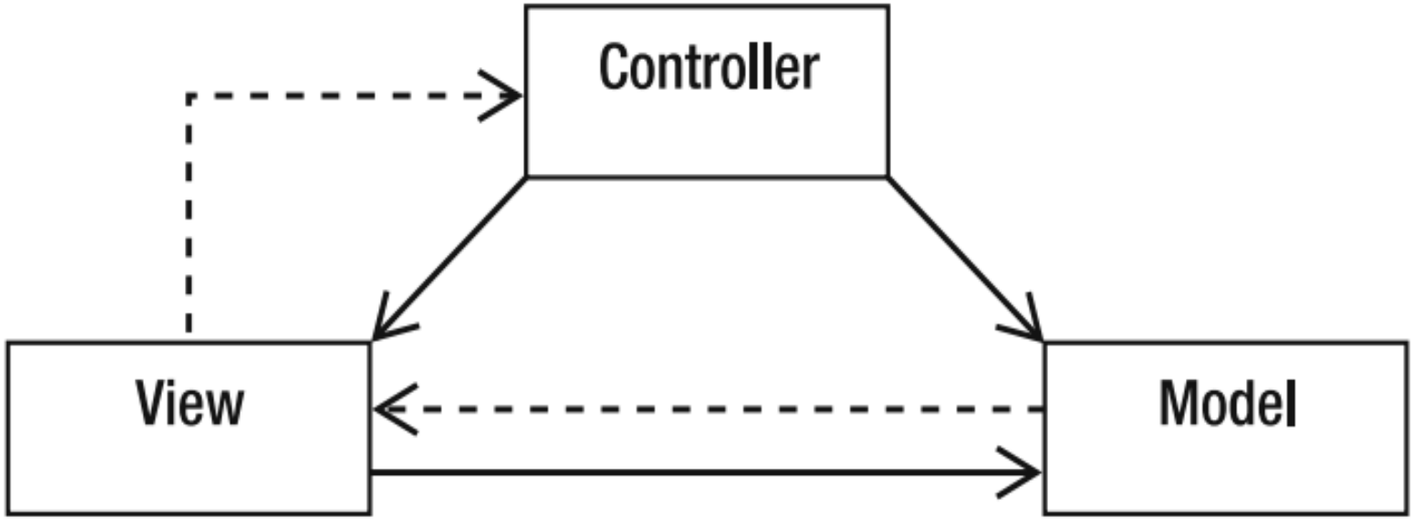


Figure 5-2. *The Model-View-Controller architecture*

The user input, the modeling of the external world, and the visual feedback to the user are separated and handled by model, view and controller *objects,* as shown in Figure 5-2.

An Object-Oriented Architectural Pattern

- The **controller** interprets mouse and keyboard inputs from the user and maps these user actions into commands that are sent to the model and/or viewport to effect the appropriate change.
- The **model** manages one or more data elements, responds to queries about its state, and responds to instructions to change state. The model knows what the application is supposed to do and is the main computational structure of the architecture – it models the problem you're trying to solve.
- The **view** or **viewport** manages a rectangular area of the display and is responsible for presenting data to the user through a combination of graphics and text. The view doesn't know anything about what the program is actually doing; all it does is take instructions from the controller and data from the model and displays them. It communicates back to the model and controller to report status.

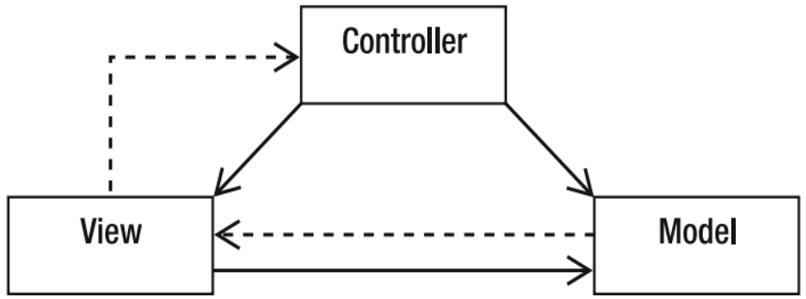


Figure 5-2. *The Model-View-Controller architecture*

An Object-Oriented Architectural Pattern

- The flow of an MVC program typically looks like this:

1. The *user* interacts with the user interface (e.g., the user presses a button) and the controller handles the input event from the user interface, often via a registered handler or callback. The user interface is displayed by the view but controlled by the controller. Oddly enough, the controller has no direct knowledge of the view as an object; it just sends messages when it needs something on the screen updated.

2. The *controller* accesses the model, possibly updating it in a way appropriate to the user's action (e.g., controller causes the user's shopping cart to be updated by the model). This usually causes a change in the model's state as well as in its data.

An Object-Oriented Architectural Pattern

3. A *view* uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view. It just responds to requests for data from whomever and to requests for transforming data from the controller.

4. The controller, as the user interface manager, waits for further user interactions, which begins the cycle anew.

An Object-Oriented Architectural Pattern

- The main idea here is separation of concerns – and code.
  - The objective is to separate how your program works from what it is displaying and how it gets its input data.
  - This is classic object-oriented programming; create objects that hide their data and hide how they manipulate their data and then just present a simple interface to the world to interact with other objects.

An MVC Example: Let's Hunt!

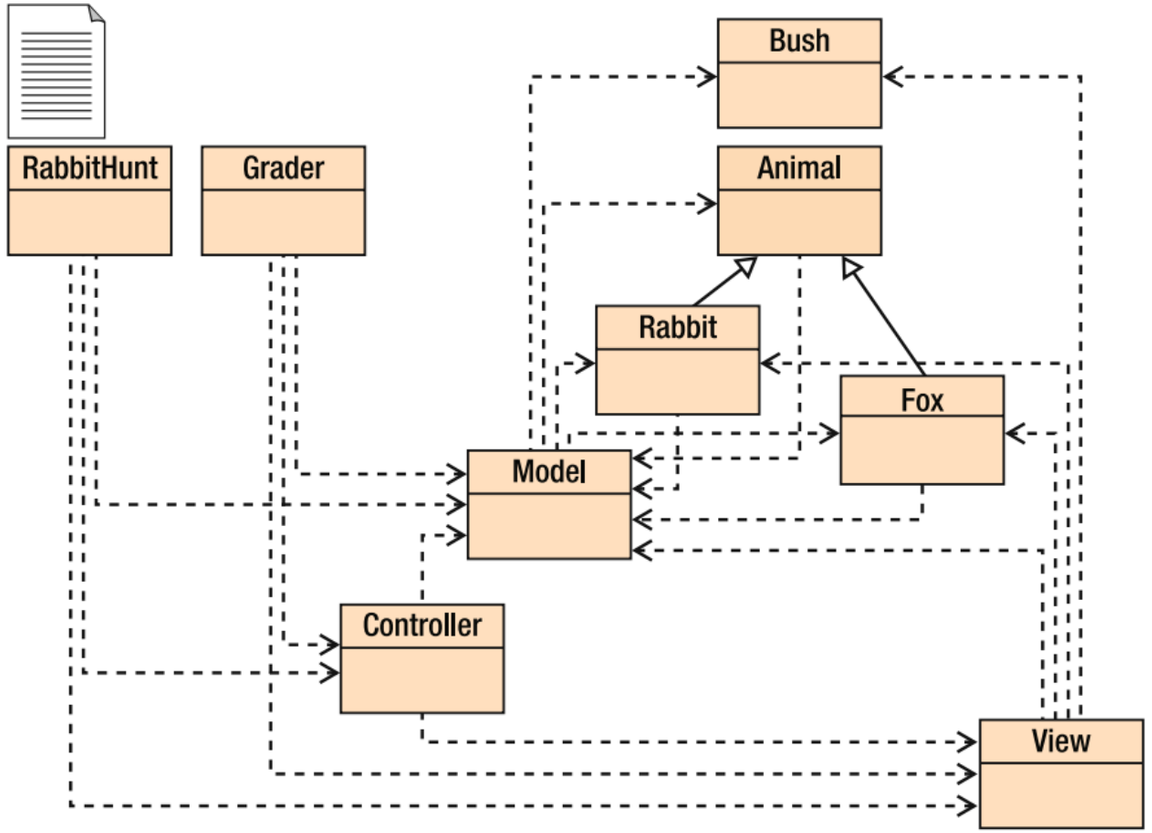


Figure 5-4. *The fox and rabbit hunt class structure*

The Client-Server Architectural Pattern

- In a client-server architecture, your program is broken up into two different pieces that typically run on two separate computers.
  - A server does most of the heavy lifting and computation; it provides services to its clients across a high-bandwidth network.
  - Clients, on the other hand, mostly just handle user input, display output, and provide communication to the server.

The Client-Server Architectural Pattern

- In short, the client program **sends requests** for services to the server program.
- The server program then evaluates the request, does whatever computation is necessary (including accessing a database, if needed) and responds to the client's request with an answer.
- The most common example of a client-server architecture today is the **World Wide Web**.

The Client-Server Architectural Pattern

- In the web model, your **browser** is the client. It presents a user interface to you, communicates with a web server, and renders the resulting web pages to your screen.
- The web server does several things. It serves web pages in HTML, but it also can serve as a database server, a file server, and a computational server.
- Clients and servers **don’t have** to be on different computers.

The Layered Approach

- The **layered architectural** approach suggests that programs can be structured as a series of layers with a sequence of well-defined interfaces between the layers.
- This has the effect of **isolating** each layer from the ones above and below it, so that one can change the internals of any layer without having to change any of the other layers in the program.
- Two classic examples of a layered approach to programming are operating systems (OSs) and communications protocols.

The Layered Approach

- In this layered model, user applications request operating system services via **a system call interface**.
- This is normally the only way for applications to access the computer's hardware.
- Most operating system services must make requests through the kernel and all hardware requests must go through device drivers that talk directly to the hardware devices.
- Each of these layers has a well-defined interface, so that, for example, a developer may add a new device driver for a new disk drive without changing any other part of the OS. This is a nice example of **information hiding**.

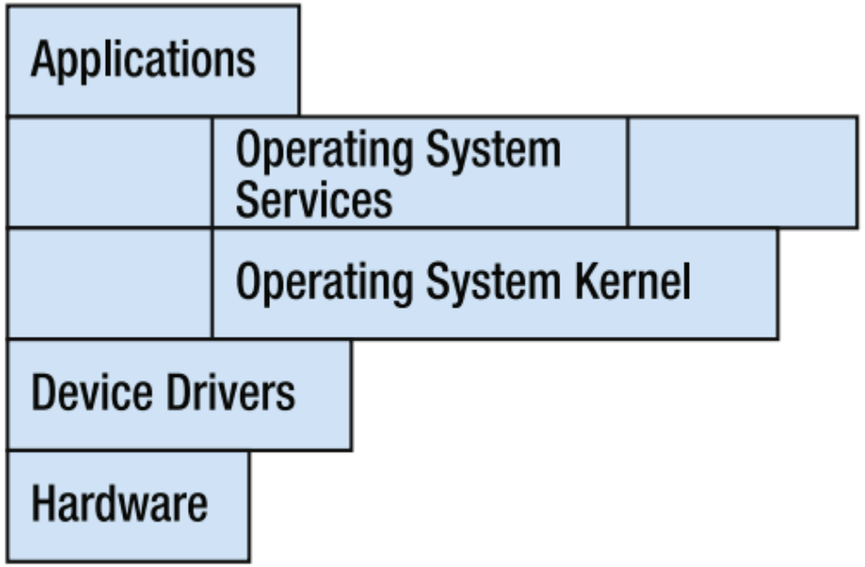


Figure 5-5. A layered architecture

The Layered Approach

- In this model, each layer contains functions or services that are logically similar and are grouped together.
- An interface is defined between each layer and communication between layers is only allowed via the interfaces.

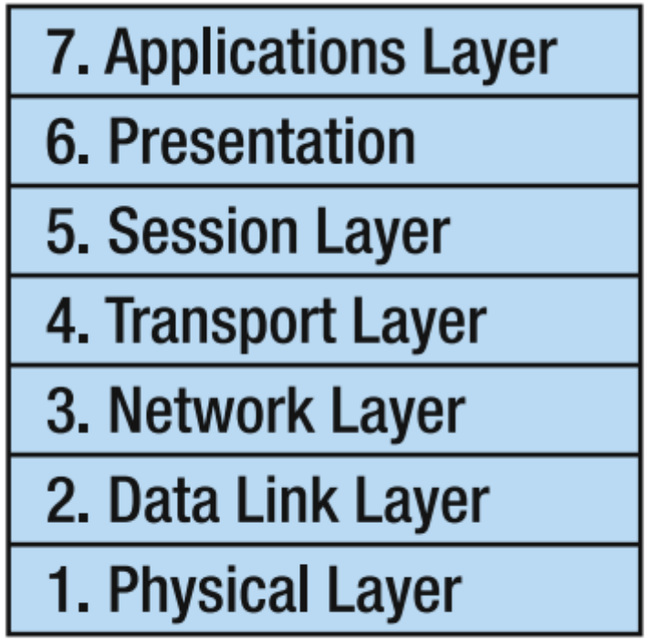


Figure 5-6. *The ISO-OSI layered architecture*

The Main Program: Subroutine Architectural Pattern

- The idea is simple. You start with a big problem, and then try to decompose the problem into several smaller problems or pieces of the original problem.
- Once you have a problem divided into several pieces, you look at each piece individually and continue dividing, ignoring all the other pieces as you go.
- Eventually, you'll have a very small problem where the solution is obvious; now is the time to write code. So you generally solve the problem from the top down and write the code from the bottom up.

The Main Program: Subroutine Architectural Pattern

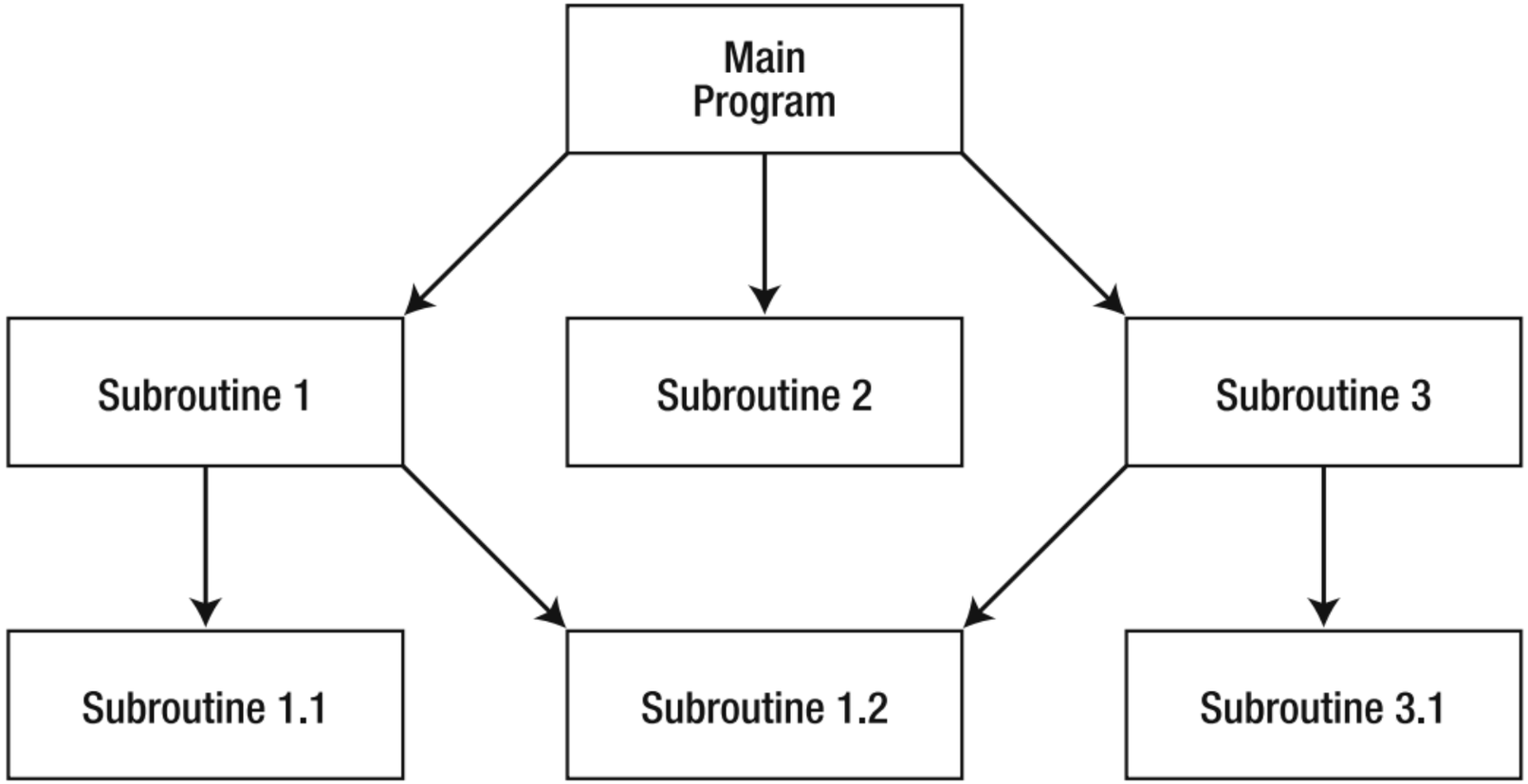


Figure 5-7. A main program – subroutine architecture

Conclusion

- The software architecture is the core of your application. It is the foundation on which you build the rest of the program. It drives the rest of your design.
- There are many different styles of software architecture and in any given project you'll probably use more than one.
- The architectural style used for a program depends on what it is you're doing.
- For software – design follows architecture.

Software Development and Professional Practice

Lecture 7: Structured Design

For
Faculty of Computers and Information
Assiut University

By
Dr. Mostafa Salem
Assistant Professor, FCI-Assiut University
https://mostafasalem.netlify.app

Structured Programming

- Problem-solving was taught in a top-down structured manner, where one begins with the problem statement and attempts to break the problem down into a set of solvable sub-problems.
- The process continues until each sub-problem is small enough to be either trivial or very easy to solve.
- This technique is called **structured programming**.
- Before the advent and acceptance of object-oriented programming in the mid-80s, this was the standard approach to problem solving and programming.
- It is still one of the best ways to approach a large class of problems.

Stepwise Refinement

- **Stepwise refinement** contends that designing programs consists of a set of refinement steps.
  - In each step, a given task is broken up into a number of subtasks.
  - Each refinement of a task must be accompanied by a refinement of the data description and the interface.
  - The degree of modularity obtained will determine the **ease or difficulty** with which a program can be adapted to changes in requirements or environment.
- During refinement, you use a notation that is natural to the problem space.
- Avoid using a programming language for description as long as possible.
- Each refinement implies several design decisions based on a set of design criteria. These criteria include efficiency of time and space, clarity, and regularity of structure (simplicity).

Stepwise Refinement

- Refinement can proceed in two ways, top-down or bottom-up.
  - **Top-down refinement** is characterized by moving from a general description of the problem to detailed statements of what individual modules or routines do.
  - Analyzing the problem and trying to identify the outlines of a solution and the pros and cons of each possibility;
  - then, designing the top levels first;
  - steering clear of language-specific details;
  - pushing down the details until you get to the lower levels;
  - formalizing each level;
  - verifying each level; and
  - moving to the next lower level to make the next set of refinements.
  - (That is, repeat.)

Stepwise Refinement

- One continues to refine the solution until it seems as if it would be easier to code than to decompose;
- That is, you work until you become impatient at how obvious and easy the design becomes.
- The down-side here is that you really have no good metric on “when to stop.” It just takes practice.

Stepwise Refinement

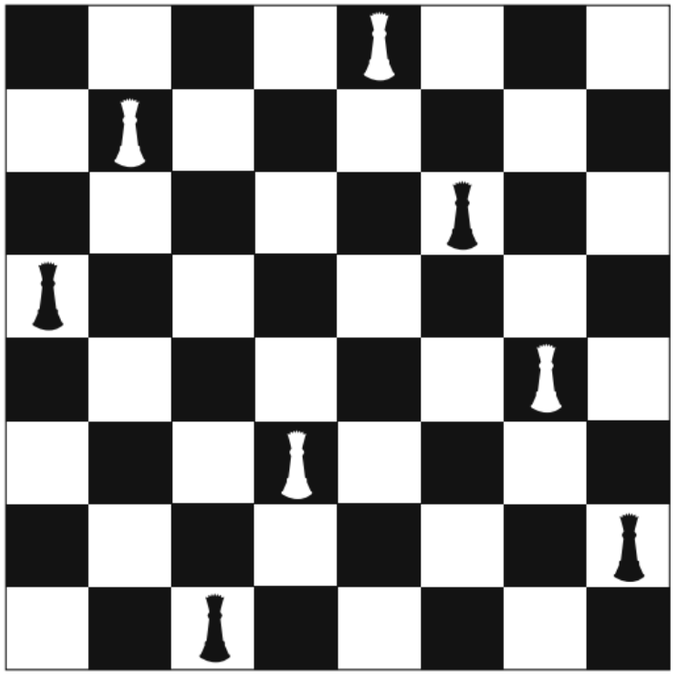
- If you can't get started at the **top**, then start at the **bottom**.
  - Ask yourself, "What do I know that the system needs to do?" This usually involves lower level I/O operations, other low-level operations on data structures, and so on.
  - Identify as many low-level functions and components as you can from that question.
  - Identify **common** aspects of the low-level components and group them together.
  - Continue with the next level up or go back to the top and try again to work down.

Stepwise Refinement

- Bottom-up assessment usually results in **early identification of utility routines**, which can lead to a more compact design.
- It also helps **promote reuse** – because you are reusing the lower-level routines.
- **On the downside**, bottom-up assessment is hard to use exclusively – you nearly always end up switching to a top-down approach at some point,
- Most real step-wise refinements involve alternating between top-down and bottom-up design elements.
- Fortunately, top-down and bottom-up design methodologies can be very complementary.

Example of Stepwise Refinement: The Eight-Queens Problem

- The eight queens' problem is familiar to most students. The problem is to find a placement of eight queens on a standard 8x8 chess board in such a way that no queen can be attacked by any other. One possible solution to the problem is shown in the following Figure.



- How would you approach this problem?

Stepwise Refinement

- Proposed Solution 1
- Proposed Solution 2
- Proposed Solution 3

Modular Decomposition

- In computer science, separation of concerns is the process of separating a computer program into **distinct features** that overlap in functionality **as little as possible**.
- A concern is any piece of interest or focus on a program.
- Progress towards separation of concerns is traditionally achieved through modularity of programming and encapsulation (or “transparency” of operation), with the help of information hiding.
- Traditionally, separation of concerns was all about separating functionality of the program.
- After that, the idea of separating the data was added as well, so that individual modules would control data as well as the operations that acted on the data and the data would be visible only through **well-defined interfaces**.

Modular Decomposition

- There are **three characteristics of modularity** that are key to creating modular programs:
  - Encapsulation
  - Loose coupling (how closely do modules relate to each other)
  - Information hiding

Modular Decomposition

- **Encapsulation** means to bundle a group of services defined by their data and behaviors together as a module and keep them together.
  - This group of services should be coherent and clearly belong together. (Like a function, a module should do just one thing.)
  - The module then presents an interface to the user and that interface is ideally the only way to access the services and data in the module.
- An objective of encapsulating services and data is high **cohesion**.
- This means that your module should do one thing and all the functions inside the module should work towards making that one thing happen.
- The closer you are to this goal, the higher the **cohesion** in your module. This is a good thing.

Modular Decomposition

- The complement of encapsulation is **loose coupling**.
  - Loose coupling describes how strongly two modules are related to each other.
  - This means we want to minimize the dependence any one module has on another.
  - We separate modules to minimize interactions and make all interactions between modules through the module interface.
  - The goal is to create modules with internal integrity (strong cohesion) and small, few, direct, visible, and flexible connections to other modules (loose coupling).
- Good coupling between modules is loose enough that one module can easily be called by others.

Modular Decomposition

- Coupling falls into **four broad categories** that go from good to awful:
  - **Simple data coupling:** Where non-structured data is passed via parameter lists. This is the best kind of coupling, because it lets the receiving module structure the data as it sees fit and it allows the receiving module to decide what to do with the data.
  - **Structured data coupling:** Where structured data is passed via parameter lists. This is also a good kind of coupling, because the sending module keeps control of the data formats and the receiving module gets to do what it wants to with the data.
  - **Control coupling:** Where data from module A is passed to module B and the content of the data tells module B what to do. This is not a good form of coupling; A and B are too closely coupled in this case because module A is controlling how functions in module B will execute.
  - **Global-data coupling:** Where the two modules make use of the same global data. This is just awful. It violates a basic tenet of encapsulation by having the modules share data. This invites unwanted side effects and ensures that at any given moment during the execution of the program that neither module A nor module B will know precisely what is in the globally shared data. And what the heck are you doing using global variables anyway? Bad programmer!

Modular Decomposition

- Information hiding is often confused with encapsulation, but they are not the same thing.
  - Encapsulation describes a process of wrapping both data and behaviors into a single entity – in our case, a module.
  - Data can be publicly visible from within a module, and thus not hidden. Information hiding,
  - On the other hand, says that the data and behaviors in a module should be controlled and visible only to the operations that act on the data within the module, so it's invisible to other, external, modules.
  - This is an important feature of modules (and later of objects as well) because it leaves control of data in the module that understands best how to manipulate the data and it protects the data from side effects that can arise from other modules reaching in and tweaking the data.

Software Development and Professional Practice

Lecture 8: Object-Oriented Analysis and Design

For
Faculty of Computers and Information
Assiut University

By
Dr. Mostafa Salem
Assistant Professor, FCI-Assiut University
https://mostafasalem.netlify.app

Object

- The object has three properties, which makes it a simple, yet powerful model building block.
  - It has state so it can model memory.
  - It has behavior, so that it can model dynamic processes.
  - And it is encapsulated, so that it can hide complexity.
- First of all, objects are things.
  - They have an identity (i.e., a name), a state (i.e., a set of attributes that describes the current data stored inside the object), and a defined set of operations that operate on that state.
  - A stack is an object, as is an Automobile, a Bank Account, a Window, or a Button in a graphical user interface.

Object-oriented program

- In an object-oriented program, a set of cooperating objects pass messages among themselves.
- The messages make requests of the destination objects to invoke methods that either perform operations on their data (thus changing the state of the object), or to report on the current state of the object.
- Eventually work gets done.
- Objects use encapsulation and information hiding (remember, they're different) to isolate data and operations from other objects in the program.
  - Shared data areas are (usually) eliminated.
  - Objects are members of classes that define attribute types and operations.

Classes

- Classes are templates for objects. Classes can also be thought of as factories that generate objects.
  - So, an Automobile class will generate instances of autos, a Stack class will create a new stack object, and a Queue class will create a new queue.
  - Classes may inherit attributes and behaviors from other classes.
  - Classes may be arranged in a class hierarchy where one class (a super class or base class) is a generalization of one or more other classes (sub-classes).
  - A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own.
  - In this sense a subclass is **more specific** and detailed than its super class; hence, we say that a sub-class extends a super class.
    - For example, a priority queue is a more specific version of a queue; it has all the attributes and operations of a queue, but it adds the idea that some queue elements are more important that others.
  - In Java this feature is called inheritance while in UML it's called generalization.

Inheritance

- There are several advantages to inheritance.
  - It is an abstraction mechanism which may be used to classify entities. It is a reuse mechanism at both the design and the programming level.
- And, of course, there are problems with inheritance, as well.
  - It makes object classes that are not self-contained; sub-classes cannot be understood without reference to their super classes.
  - Inheritance introduces complexity and this is undesirable, especially in critical systems.

OOP

- Object-oriented programming (OOP) has several advantages, among them
  - Easier maintenance, because objects can be understood as stand-alone entities.
  - Objects are also appropriate as reusable components.
- But, for some problems there may be no mapping from real-world objects to system objects, meaning that OOP is not appropriate for all problems.

An Object-Oriented Analysis and Design Process

- Object-oriented analysis (OOA), design (OOD) and programming (OOP) are related but distinct.
  - **OOA** is concerned with developing an object model of the application domain.
    - So, for example, you take the problem statement, generate a set of features and (possibly) use cases, tease out the objects and some of the methods within those objects that you'll need to satisfy the use case, and you put together an architecture of how the solution will hang together.
    - That's object-oriented analysis.

An Object-Oriented Analysis and Design Process

- Object-oriented analysis (OOA), design (OOD) and programming (OOP) are related but distinct.
  - **OOD** is concerned with developing an object-oriented system model to satisfy requirements.
    - You take the objects generated from your OOA, figure out whether to use inheritance, aggregation, composition, abstract classes, interfaces, and so on, in order to create a coherent and efficient model, draw the class diagrams, and flesh out the details of what each attribute is and what each method does, and describe the interfaces.
    - That's the design.

1. Write (or receive) the problem statement. Use this to generate an initial set of features.
2. Create the feature list. The feature list is the set of program features that you derive from the problem statement; it contains your initial set of requirements.
3. Write up use cases. This helps to refine the features and to dig out new requirements and to expose problems with the features you just created. We'll also see that we can use user stories for this step.
4. Break the problem into subsystems or modules or whatever you want to call them as long as they're smaller, self-contained bits usually related to functionality.
5. Map your features, subsystems, and use cases to domain objects; create abstractions.
6. Identify the program's objects, methods, and algorithms.
7. Implement this iteration.
8. Test the iteration.
9. If you've not finished the feature list and you still have time and/or money left, go back to step 4 and do another iteration, otherwise…
10. Do final acceptance testing and release.

OOP

- How do the process steps above fit into the software development life cycle?
- Recall that the basic development life cycle has four steps:
  - 1. Requirements Gathering and Analysis;
  - 2. Design;
  - 3. Implementation and Testing; and
  - 4. Release, Maintenance, and Evolution.

OOP

Requirements Gathering and Analysis

1. Problem statement.
2. Feature list creation.
3. Use case generation.

Design

1. Break up the problem.
2. Map features and use cases to domain objects.
3. Identify objects, methods, and algorithms.

Implementation and Testing

1. Implement this iteration.
2. Test the iteration.
3. If you've not finished with the feature list or out of time, go back to step 4, otherwise…

Release/Maintenance/Evolution

1. Do final acceptance testing and release.

Doing the Process

- Let’s continue by working through an extended example, seeing where the problem statement leads us and how we can tease out requirements and begin our object-oriented analysis.

The Problem Statement

Burt, the proud owner of Birds by Burt, has created the ultimate in bird feeders. Burt's Bird Buffet and Bath (B^4), is an integrated bird feeder and bird bath. It comes in 12 different colors (including camo) and 1, 3, and 5 lb capacities. It will hold up to one gallon of water in the attached bird bath, it has a built-in hanger so you can hang it from a tree branch or from a pole, and the B^4 is just flying off the shelves. Alice and Bob are desperate for a B^4, but they'd like a few changes. Alice is a techno-nerd and a fanatic songbird watcher. She knows that her favorite songbirds only feed during the day, so she wants a custom B^4 that allows the feeding doors to open automatically at sunrise and close automatically at sunset. Burt, ever the accommodating owner, has agreed and the hardware division of Birds by Burt is hard at work designing the B^4++ for Alice. Your job is to write the software to make the hardware work.

The Feature List

- The first thing we need to do is figure out what the B4++ will actually do.
- We can almost immediately write down three requirements:
  - The feeding doors must all open and close simultaneously.
  - The feeding doors should open automatically at sunrise.
  - The feeding doors should close automatically at sunset.
- The next step is to create a use case so we can see just what the bird feeder is really going to do.

Use Cases

- A use case is a description of what a program does in a particular situation.
- It's the detailed set of steps that the program executes when a user asks for something.
- Use cases always have an actor – some outside agent that gets the ball rolling, and a goal – what the use case is supposed to have done by the end.
- The use case describes what it takes to get from some initial state to the goal from the user's perspective.

Use Cases

- Here's a quick example of a use case for the B4++:
  - 1. The sensor detects sunlight at a 40% brightness level.
  - 2. The feeding doors open.
  - 3. Birds arrive, eat, and drink.
  - 4. Birds leave.
  - 5. The sensor detects a decrease in sunlight to a 25% brightness level.
  - 6. The feeding doors close.

Use Cases

- Note that in the use case we don’t talk about how a program does something, we only concentrate on what the program has to do to reach the goal.
- Use cases are generated during the Requirements Gathering and Analysis phase of the software life cycle, so we’re not so much concerned with the details yet, we just treat the program as a black box and let the use case talk about the external behavior of the program.

Decompose the Problem

- So now that we've got our use case we can probably just decompose the problem and identify the objects in the program.
- This problem is quite simple; if you look at the use case above and pick out the nouns, you see that we can identify several objects. Each of these objects has certain characteristics and contributes to reaching the goal of getting the birds fed.
- The other two nouns of interest are "sensor" and "doors." These are the critical pieces of the B4++, because the use case indicates that they are the parts that accomplish the goal of opening and closing the feeding doors at sunrise and sunset. So, it's logical that they are objects in our design.

Decompose the Problem

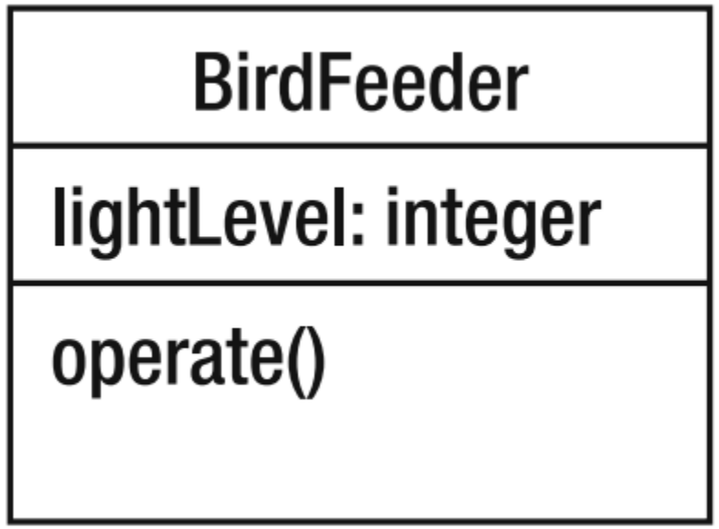
- Here are the objects I came up with for this first version of the B4++ and a short description:
  - **BirdFeeder:** The top-level object. The bird feeder has one or more feeding doors at which the birds will gather, and a sensor to detect sunrise and sunset. The BirdFeeder class needs to control the querying of the light sensor and the opening and closing of the feeding doors.
  - **Sensor:** There will be a hardware light sensor that detects different light levels. We'll need to ask it about light levels.
  - **FeedingDoor:** There will be several feeding doors on the bird feeder. They have to open and close.
- To describe classes and their components we can use another UML feature, class diagrams.

Class Diagrams

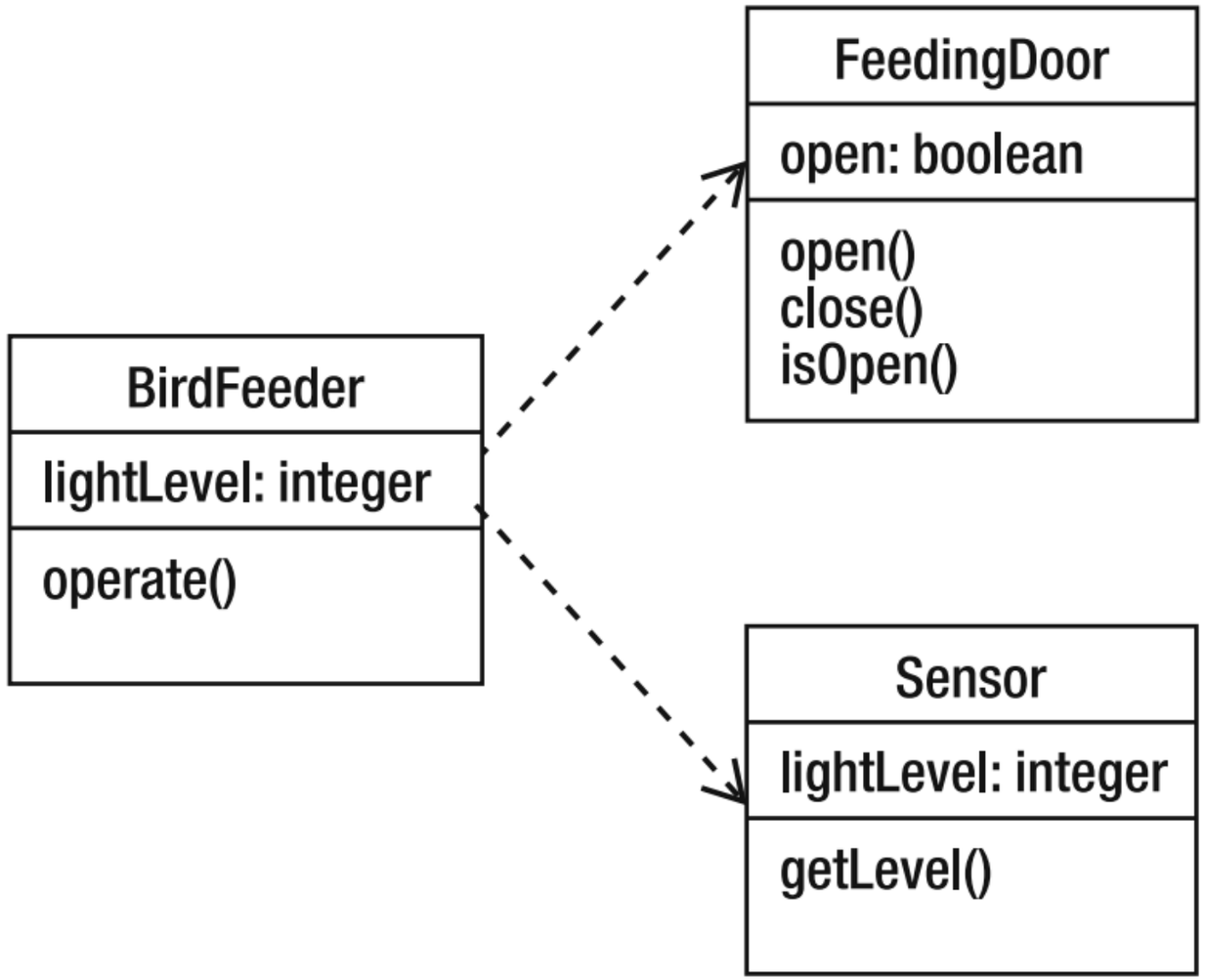
- A class diagram allows you to describe the attributes and the methods of a class.
- A set of class diagrams will describe all the objects in a program and the relationships between the objects.
- We draw arrows of different types between class diagrams to describe the relationships.
- Class diagrams give you a visual description of the object model that you've created for your program.

Class Diagrams

- Class diagrams have three sections:
  - **Name:** The name of the class
  - **Attributes:** The instance data fields, and their types used by the class
  - **Methods:** The set of methods used by the class and their visibility.



Class Diagrams



In UML, the dotted line with the open arrow at the end indicates that one class (in our case BirdFeeder) is *associated* with another class (in our case either FeedingDoor or Sensor) by *using* it.

Code

- In the BirdFeeder object, the operate() method needs to check the light levels and open or close the feeding doors depending on the current light level reported by the Sensor object, and does nothing if the current light level is above or below the threshold values..
- In the Sensor object, the getLevel() method just reports back the current level from the hardware sensor.
- In the FeedingDoor object, the open() method checks to see if the doors are closed. If they are, it opens them and sets a boolean to indicate that they're open. The close() method does the reverse.