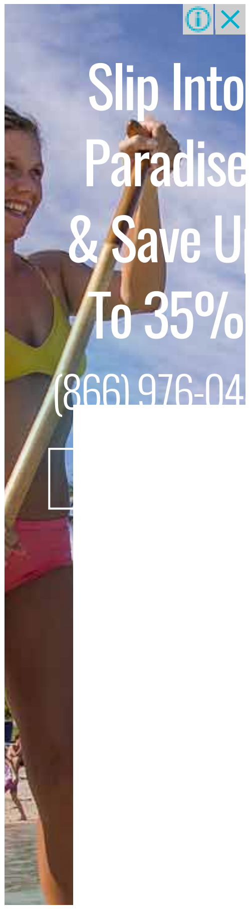
Software Engineering Tips

Search this site



Home >

How to design a computer program

This tutorial assumes you're designing a standalone computer program that runs with a conventional GUI or command-line interface, but many of the techniques can also apply to programs that will become part of a bigger system. It's the next in a series after "So you've just been given your first programming assignment" and expands on what goes on during Step 7 of that tutorial: **Sketching a plan of attack**. It'll introduce you to the most important principles of program design and gloss over some illustrative examples and specifics.

A standalone program is one that is justified all by itself, like a word processor or a game, but even if it was a cog in a bigger system it'd still have the same qualities: it would focus on one job, it would take some kind of input that the system produces and transform it into some kind of output that the user or system consumes. Some examples of good standalone programs are:

- Address book
- Spreadsheet
- Calculator
- Trip planner
- Picture editor

Whereas a program designed to be part of a larger system can be something like:

- A program that imports purchase orders and saves them to a database
- A program that prints packing slips for orders stored in a database
- A web browser
- An email client
- An MP3 player

The first set of programs are all useful on their own, but the second set all need something else to complete them such as a web server or email server. Even the MP3 player needs a program that can create new MP3 files for it to play. The impact to you is that the second set of programs have to cooperate with some kind of protocol or file format that you didn't design yourself, and so they become part of your spec. In this tutorial I'm going to ignore that aspect so I can concentrate on the basics.

What is the difference between designing and coding?

There is no difference between designing a program on paper and coding it, but code tends to be harder to understand and change. So the goal of the paper planning phase is to invent a pattern that helps you understand the code and isolate it from changes made in other parts of the program. In fact, **most of your time will be spent creating ways to isolate code from changes made to other code**, and most of this tutorial will be about how you can do this one thing.

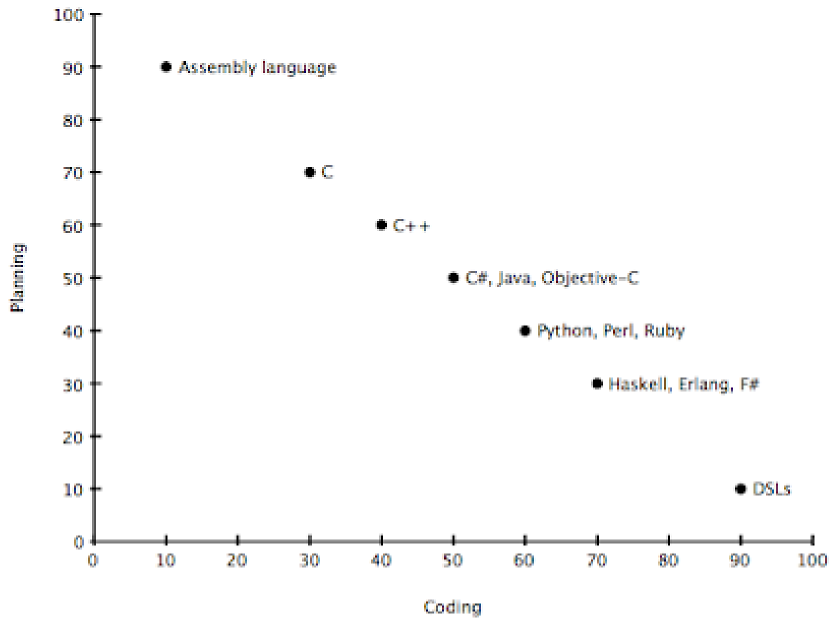
A popular metaphor for design versus coding is the architect who gives blueprints to the builder, and the builder represents the programmer. But it's an imperfect metaphor because it forgets the compiler, which does the same thing as the builder in the architect metaphor. If we revised the metaphor then the architect would have to go through three phases to design a bridge: come up

ترجمة

to organize the blueprints, then come up with a way to prevent changes in the design of the bolts or girders from forcing a redesign of the entire bridge, and *then* design the bridge.

In real life construction projects the builders also co-design the bridge along with the architect because they discover problems that the architect missed, even offering solutions for them based on their expertise as builders. Not just problems with the measurements and materials either, but problems with the landscape, weather and budget. With programming the compiler will be a co-designer because it'll also tell you about problems you missed. Not just problems with the syntax, but problems with the way you use *types*. Many modern compilers will go as far as to suggest ways to fix those problems based on the expertise of the programmers who made it.

This is why the choice of language has a big impact on how much design work you'll do on paper before you begin coding. The more type checking and static analysis the compiler does and the more expressive the language is then the easier it is to organize and isolate code. Imagine that languages could be placed on a graph like the one below. Here we see that assembly languages require the most planning in advance of coding, but DSLs (Domain Specific Languages) can be so specialized for their task that you can jump into coding almost right away.



The numbers on this graph have been exaggerated to illustrate the idea, but the gist is that every language aims to sit somewhere between intense planning and carefree coding. And even though it seems like it would be nice to operate at the bottom-right corner all the time, it isn't necessarily an advantage; one of the trade-offs of expressibility in any language--human or computer languages--is applicability. Every abstraction comes at a price that makes the language less appropriate for some other task.

What makes a language "expressive"?

The more work you can imply with a single symbol or keyword, and the more open a symbol is to modification by other symbols, then the more expressive the language is.

For example, the loop. First we had to define loops by initializing a counter, then establishing a label for the beginning of the loop, and then increment the counter, test for a condition, and GOTO the beginning of the loop again:

```
10 DIM i = 1
20 PRINT i
30 i = i + 1
40 IF i < 10 GOTO 20
```

Then languages got to be a little bit more expressive, and it took less work to say the same thing:

```
for (int i = 0; i < 10; i++)
    print(i);
```

The new language added symbols and syntax that encapsulated more meaning, not just making it easier to see that it was a loop, but even the syntax for incrementing the counter got shorter.

These were symbols that implied a lot of meaning, but there's also the ability to modify the meaning of symbols:

Tomorrow = Now + "24 hours"

Not only was the plus operator overloaded to perform the appropriate arithmetic on dates, but implicit conversion was invoked to convert the string literal into a timespan value.

Even if your problem lives at the bullseye of a language's target there will still never be a language so expressive that you'll never need any kind of paper-planning. There will only ever be languages that make it safer to move into coding *earlier*.

How to think about the design of a program

All programs transform input into output, even if that input was hard-coded into the source at design-time. It's obvious enough to be forgotten and programmers can get so lost in the fuss of implementing features that they forget what the real purpose of the program was. Proper design can help you and your team mates avoid losing sight of the real goal.

The code that does the conversion is called the **business logic**. The business logic will act on data structures called the **model**. Your program will need to convert (or "deserialize") its input into the model, transform it, and then serialize the model into output. The model will be whatever makes it easiest to perform the transformation and does not need to resemble what the user thinks of the data as. For example, a calculator converts decimal numbers into binary, performs arithmetic on them there, and then converts the binary back into decimal. The binary representation of the decimal numbers is the calculator's model.

Everything else is called **overhead**, and that will be code that implements the user interface, sanitizes the input, creates connections to services, allocates and disposes of memory, implements higher-level data structures like trees and lists and so-on. One of your goals will be to separate the overhead from the business logic so that you can make changes to one without having to modify the other.

Some mistakes to avoid at this stage:

1. Don't create a class or module called "BusinessLogic" or similar. It shouldn't be a black-box. Later I'll discuss how to structure this code
2. Don't put business logic in a library[1]. You'll want the ability to change it more often than the overhead, and libraries tend to get linked to other programs--multiplying the consequences of changing it
3. Don't put business logic or overhead into the model unless it's only enforcing constraints. The code that defines the model should expose some interfaces that let you attach what you need. More on that later

Some of the things that you can include in the model is code that enforces meaningful relationships. So if your genology program has a "Person" class with an "Offspring" collection then you should include code in the model that prevents a person from being their own grandparent. But the function to discover cousin relationships in a family tree is *business logic* and doesn't belong in the model. Although it might make sense that you can't be your own cousin, and that this rule could be enforced in the model, you also need to draw a line between what enforcement is necessary to **prevent common bugs** and what's going to **create complex bugs** through mazes of logic that are difficult to follow.

1 - "Anything Is Appropriate As Long As You Know What You're Doing." One man's business logic is another man's overhead, and your project might be packaged as a library to be consumed by a bigger system. This rule is meant to apply to the point you're at in the food chain.

Top-Down versus Bottom-Up

Top-Down programming is about writing the business logic first in pseudocode or non-compiling code and then working down to the specifics, and it's the first program design technique taught in schools. Lets say that you're designing a coin-counting program, and you write what you want the perfect main() to look like:

```
void main()
{
    CoinHopper hopper = new CoinHopper(Config.CoinHopperHardwarePort);
    decimal deposit = 0;
    while (hopper.count > 0)
    {
        CoinType coin = hopper.next();
        deposit += coin.value;
    }

    decimal processingFee = deposit * 0.08;
    Output.Write("Total deposit: ${0}, processing fee: ${1}, net deposit: ${2}",
                 deposit,
                 processingFee,
                 deposit - processingFee);
}
```

The program doesn't compile yet, but you've now **defined the essence of the program in one place**. With top-down programming you start like this and then you focus on defining what's going on in the `CoinHopper` class and what abstraction is going on in the `Output` class.

Bottom-up programming is the natural opposite, where the programmer intuits that he'll need a way to control the coin counting hardware and begins writing the code for that before writing the code for main(). One of the reasons for doing it this way is to discover non-obvious requirements that have a major impact on the high-level design of the program. Lets say that when we dig into the nitty-gritty of the API for the hardware we discover that it doesn't tell us what type of coin is there, it just gives us sensory readings like the diameter and weight. Now we need to change the design of the program to include a module that identifies coins by their measurements.

What usually happens in the real-world is that all of the top-down design is done on paper and the actual coding goes bottom-up. Then when the coding effort falsifies an assumption made on paper the high-level design is easier to change.

> "A further disadvantage of the top-down method is that, if an understanding of a fault is obtained, a simple fix, such as a new shape for the turbine housing, may be impossible to implement without a redesign of the entire engine." - from Richard Feynman's report of the Space Shuttle Challenger Disaster

High-level design patterns

"Design patterns" might have come into the programmer's lexicon around the time a book by the same name was published, but the concept has existed since the dawn of man. Given a problem, there are certain traditional ways of solving it. Having windows on two adjacent walls in a room is a design pattern for houses that ensures adequate light at any time of day. Or putting the courthouse on one side of a square park and shops around the other sides is another kind of design pattern that focuses a town's community. The culture of computer programming has developed hundreds of its own design patterns: high level patterns that affect the architecture of the whole program--forcing you to chose them in advance--and low-level patterns that can be chosen later in development.

High-level patterns are baked into certain types of frameworks. The **Model-View-Controller** (MVC) pattern is ubiquitous for GUIs and web applications and nearly impossible to escape in frameworks like Cocoa, Ruby-on-Rails, and .Net, but that doesn't mean you have to use it for every program; some programs don't have to support random user interaction and wouldn't benefit from MVC.

MVC is well described elsewhere, so here are some of the other high-level patterns and what they're good for.

The 1-2-3

This is the first pattern you learned in programming class: the program starts, asks the user for input, does something with it, prints it out and halts. One-two-three. "Hello world" is a 1-2-3. It's the simplest pattern and still useful and appropriate in many situations. It's good for utilities, housekeeping tasks and one-off programs.

One of the biggest mistakes made by programmers is to get too ambitious. They start designing an MVC program, take weeks or months to code and debug it, but then it *dwarfs the magnitude of the problem*. If someone wants a program that doesn't need to take its inputs randomly and interactively, then produces a straightforward output, then it's a candidate for 1-2-3.

The Read-Execute-Print Loop (REPL)

REPL is the 1-2-3 kicked up a notch. Here the program doesn't halt after printing its output, it just goes back and asks for more input. Command-line interfaces are REPLs, and most interpreted programming languages come with a REPL wrapped around the interpreter. But if you give the REPL the ability to **maintain state** between each command then you have a powerful base to build simple software that can do complex things. You don't even need to implement a full programming language, just a simple "KEYWORD {PARAMETER}" syntax can still be effective for a lot of applications.

The design of a REPL program should keep the session's state separate from the code that interprets commands, and the interpreter should be agnostic to the fact that it's embedded in a REPL so that you can reuse it in other patterns. If you're using OOP then you should also create an Interface to abstract the output classes with, so that you aren't writing to STDOUT directly but to something like `myDevice.Write(results)`. This is so you can easily adapt the program to work on terminals, GUIs, web interfaces and so-on.

The Pipeline

Input is transformed one stage at a time in a pipeline that runs continuously. It's good for problems that transform large amounts of data with little or no user interaction, such as consolidating records from multiple sources into a uniform store or producing filtered lists according to frequently changing rules. A news aggregator that reads multiple feed standards (like RSS and Atom), filters out dupes and previously read articles, categorizes and then distributes the articles into folders is a good candidate for the Pipeline pattern, especially if the user is expected to reconfigure or change the order of the stages.

You write your program as a series of classes or modules that share the same model and have a loop at their core. The business logic goes inside the loop and treats each chunk of data atomically--that is, independent of any other chunk. If you're writing your program on Unix then you can take advantage of the pipelineing system already present and write your program as a single module that reads from stdin and writes to stdout.

Some language features that are useful for the Pipeline pattern are coroutines (created with the "yield return" statement) and immutable data types that are safe to stream.

The Workflow

This is similar to the Pipeline pattern but performs each stage through to completion before moving onto the next. It's good for when the underlying data type doesn't support streaming, when the chunks of data can't be processed independently of the others, when a high degree of user interaction is expected on each stage, or if the flow of data ever needs to go in reverse. The Workflow pattern is ideal for things like processing purchase orders, responding to trouble tickets, scheduling events, "Wizards", filing taxes, etc..

You'd wrap your model in a "Session" that gets passed from module to module. A useful language feature to have is an easy way to serialize and deserialize the contents of the session to-and-from a file on disk, like to an XML file.

The Batch Job

The user prepares the steps and parameters for a task and submits it. The task runs asynchronously and the user receives a notification when the task is done. It's good for tasks that take more than a few seconds to run, and especially good for jobs that run forever until cancelled. Google Alerts is a good example; it's like a search query that never stops searching and sends you an email whenever their news spider discovers something that matches your query. Yet even something as humble as printing a letter is also a batch job because the user will want to work on another document while your program is working on queuing it up.

Your model is a representation of the user's command and any parameters, how it should respond when the task has produced results, and what the results are. You need to encapsulate all of it into a self-contained data structure (like "`PrintJobCriteria`" and "`PrintJobResults`" classes that share nothing with any other object) and make sure your business logic has thread-safe access to any resources so it can run safely in the background.

When returning results to the user you must make sure you avoid any thread entanglement issues. Framework classes like BackgroundWorker are ideal for marshaling the results back onto the GUI.

Combining architectural patterns

You can combine high-level patterns into the same program, but you will need to have a very good idea of what the user's problem is because one model will always be more dominant than the others. If the wrong model is dominant then it'll spoil the user's interaction with the program.

For example, it wouldn't be a good idea to make the Pipeline dominant and implement 1-2-3 in one of its modules because you'll force the user to interact excessively for each chunk of data that goes through the pipeline. This is exactly the problem with the way file management is done in Windows: you start a pipeline by copying a folder full of files, but an intermediate module wants user confirmation for each file that has a conflict. It's better to make 1-2-3 dominant and let the user define in advance what should happen to exceptions in the pipeline.

User Interface patterns

A complete discussion of UI patterns would be beyond the scope of this tutorial, but there are two meta-patterns that are common between all of them: **modal** and **modeless**. Modal means the user is expected to do one thing at a time, while modeless means the user can do anything at any time. A word processor is modeless, but a "Wizard" is modal. Regardless of what the underlying architectural pattern is, the UI needs to pick one of these patterns and stick to it.

Some programs can mix the two effectively if the task lends itself, like tax-filing software because it lets you jump to points randomly in the data entry stage, but doesn't let you move onto proofing and filing until the previous stage is complete.

Almost all major desktop software is modeless and users tend to expect it. To support this kind of UI there are two mechanisms supported by most toolchains; event-driven and data binding.

Event driven

Sometimes called "call-backs", but when they're called "Events" it's usually because they've been wrapped in a better abstraction. You attach a piece of code to a GUI control, and when the user does something to the control (click it, focus it, type in it, mouse-over it, etc) your code is invoked by the control. The call-back code is called a *handler* and is passed a reference to the control and some details about the event.

Event-driven designs require a lot of code to examine the circumstances of the event and update the UI. The GUI is dumb, knows nothing about the data, needs to be spoon-fed the data, and passes the buck to your code whenever the user does something. This can be desirable when the data for the model is expensive to fetch, like when it's across the network or there's more than can fit in RAM.

Events are an important concept in business logic, overhead and models, too. The "PropertyChanged" event is very powerful when added to model classes, for example, because it fits so well with the next major UI mechanism below:

Data Binding

Instead of attaching code to a control you attach the *model* to the control and let a smart GUI read and manipulate the data directly. You can also attach `PropertyChanged` and `CollectionChanged` events to the model so that changes to the underlying data are reflected in the GUI automatically, and manipulation on the model performed by the GUI can trigger business logic to run. This technique is the most desirable if your model's data can be retrieved quickly or fit entirely in RAM.

Data binding is complemented with conventional event-driven style; like say your address-book controls are bound directly to the underlying `Address` object, but the "Save" button invokes a classic Event handler that actually persists the data.

Data binding is extremely powerful and has been taken to new levels by technologies like Windows Presentation Foundation (WPF). A substantial portion of your program's behavior can be declared in data bindings alone. For example: if you have a list-box that is bound to a collection of `Customer` objects then you can bind the detail view directly to the list-box's `SelectedItem` property and make the detail view change whenever the selection in the list-box does, no other code required. A perfectly usable viewer program can be built by doing nothing more than filling the model from the database and then passing the whole show over to the GUI. Or like MVC without the C.

What it'll mean for the design of your program is greater emphasis on the model. You'll either design a model that fits the organization of the GUI, or build adaptors that transform your model into what's easiest for the GUI to consume. The style is part of a broader concept known as Reactive Programming.

Designing models

Remember that your data model should fit the solution, not the problem. Don't look at the definition of the problem and start creating classes for every noun that you see, and nor should you necessarily create classes that model the input or output--those may be easily convertible without mapping them to anything more sophisticated than strings.

Some of the best candidates for modelling first are actions and imaginary things, like transactions, requests, tasks and commands. A program that counts coins probably doesn't need an abstract `Coin` class with `Penny`, `Nickel`, `Dime` and `Quarter` dutifully subclassing it, but it should have a Session class with properties to record the start time, end time, total coin count and total value. It might also contain a Dictionary property to keep separate counts for each type of coin, and the data type representing the coin type may just end up being an Enum or string if that's all it really needs.

Once you've modeled the actions you can step forward with your business logic a little until you begin to see which nouns should be modeled with classes. For example, if you don't need to know anything about a customer except their name, then don't create a `Customer` class, just store their name as a string property of an `Order` or `Transaction` class[2].

2 - There can be a benefit to creating "wrappers" for scalar values for the sake of strong-typing them. In this case Customer would just

store a name, but the fact that it's an instance of Customer means the value can't be accidentally assigned to a PhoneNumber property. But I wouldn't use this technique unless your language supports Generics (more below).

Organizing business logic

Earlier I said that it's bad form to create a class called BusinessLogic and cram it all in there because it makes it look like a black box. Let's say our coin-counter's business logic can be expressed as "Count all US-issue coins, discard foreign coinage and subtract an 8% processing fee." Our design has evolved since our first attempt and now we have two classes: `Tabulator` and `CoinIdentifier`.

`CoinIdentifier` has a public static method that takes input from the coin sensors in the machinery and spits out a value that identifies what kind of coin it is. It's probably going to need changes once a year or less as new coins are issued, so we have a good reason to isolate it from the rest of the code.

`Tabulator` handles a `CoinInserted` event raised by the hardware and passes the sensor data--contained within the event's arguments--to CoinIdentifier. Tabulator keeps track of the counts for all coin types regardless of whether they're US-issue or not. The essence of this code is the least likely to change, so we make sure it doesn't have to know how to identify coins or judge what ought to be done with them.

Tabulator raises a `CoinCounted` event instead, and we put its handler in the main module of the program to decide whether to keep the coin or eject it into the returns tray. The code which is the most likely to change frequently--like which coins to reject and how much to keep as a processing fee--stays in the main module, with constants like the processing fee read from a configuration file.

Organizing business logic means grouping it by purpose, avoiding bundles of unrelated functions, and creating Interfaces that let us connect the modules while insulating them from changes in each other. Another benefit is that well isolated code is easier to *test* in isolation, and that makes it safer to change other code without forcing a re-test of everything that uses it.

Keeping models, overhead and business logic separate

The last topic I'll discuss in this tutorial is also the most important, because it gives you more flexibility to change the higher-level design. The future maintainability of your code depends on it, but in large projects even the initial coding will succeed or fail on this principle.

This is what most of the book Design Patterns is about. If you buy it then be aware that some of its patterns may be obsolete in your language because they've been absorbed into language features. The "Iterator" pattern, for example, is now hidden in a language feature called list comprehensions. The pattern is still there, but now it's automatic and the language just became more *expressive*.

New patterns are being invented all the time, but these are the basic principles they all strive for:

Overhead code needs to be as program-agnostic as possible

Assume that every line of business logic and every property of the model will change and that the overhead code should expect it. You can do this by pushing that overhead code into its own classes and using **Interfaces** on models and business logic to define the *very least* of what the overhead needs to know about an object. You can also now find languages like Java, C# and Visual Basic that support **Generics**, which extend the type-checking power of the compiler (and by extension, the IDE). Here's how to make them work for you:

- **Use Interfaces to abstract an *object's* capabilities**
  If your overhead code needs to do something to an object then put an Interface on that object's class that exposes whatever is needed. For example, your overhead code has to route a message based on its destination, so the message class would implement an interface you'd call `IDeliverable` that defines a `Destination` property
- **Use Generics to abstract an *algorithm's* capabilities**
  When an algorithm doesn't care what kind of object its manipulating then it should use generics to make the algorithm transparent to the type-checking power of the compiler

Defer specifics until runtime

Near the beginning I said that every program transforms input into output, and that input will include more than just what the user types or clicks after the program starts. It's also going to include configuration files and, for the sake of clarity, you can even consider your program's main() module to be like a *compiled and linked* configuration file; its job is to assemble the abstracted pieces at the last minute and give them the kiss of life.

You can defer specifics with a design principle called **Inversion of Control** (IoC). The simplest expression is when you pass objects that know their own specifics into methods that don't, such as when a `Sort()` algorithm takes collections of objects that all implement `IComparable`. Sort() doesn't need to know how to compare two objects, it just churns through them calling `object1.Compare(object2)`. The code for Sort() concerns only what Sort() does, not what's peculiar about the things it's sorting.

Blocking events

Events can be used to invert control and pass the buck up the chain, too, and when the child code waits for the event to be handled--rather than continuing on asynchronously--we say it's a blocking event. Event names that begin with `Resolve` often want the parent code to look for something it needs. An XML parser, for example, might use a `ResolveNamespace` event rather than force the programmer to pass all possible namespaces or a namespace resolving object at the beginning. The handler has the job of finding the resource and it passes it back by setting a property in the event's arguments.

Dependency Injection

A more complex expression of IoC is called **Dependency Injection** (DI). This is when you identify a need--such as writing to different kinds of databases or output devices--and pass implementations of them into an object that uses them. Our `Tabulator` class from earlier, for example, could be passed an instance of the exact breed of `CoinIdentifier` in its constructor. We have one that works on US coins and another that works on Canadian, and a configuration file tells the `main()` which one to create.

The second method of DI is to call a **static method** from within the class that needs the resource. This is popular for logging frameworks: say your class has a member called `_logger` and in your constructor you set it by calling the static method `LogManager.GetLogger()`. GetLogger() in turn implements the Factory design pattern to pick and instantiate the appropriate kind of logger at runtime. With a tweak to a config file you can go from writing to text files to sending log events over the network or storing them in a database.

To chose between the two I like to consider how critical the dependency is to the purpose of the module. When the dependency is critical I pass it as a *parameter to the constructor* so that it's clear and explicit what's going on in your code. When it's for a side-concern like logging I hide them with calls to static methods because I want to avoid ruining the readability of the business logic.

A third method is to use a dependency injection framework like Ninject or Unity. These support more sophisticated methods for both explicit and implicit injection of the dependency, but they're only worth if if your design uses DI extensively. Manual DI has a sweet spot of around 1-to-3 injections per object while frameworks start at about 4 or more. One of the problems they bring is more configuration and prep time before they're usable, and overdosing on them can hide too much.

Parting shots

- **Don't O.D. on third-party structural libraries**
  There's you, the language's vendor, and Bob who has this wicked awesome Dependency Injection framework and Monad library. One... maybe two Bob's Frameworks are okay, but don't overdose on them or even you won't know how your program works
- **Brevity is the soul of an Interface**
  When you design interfaces for your classes to abstract their capabilities, remember to keep them short. It's normal for an Interface to specify only one or two methods and for classes to implement 5 or 6 Interfaces
- **Explain your program through its structure**
  See if you can group your functions into classes and name those classes so that when you look at the project tree you can sense what the program does before you even look at the code
- **Embrace messiness... and refactoring**
  During development (and deadlines) it's normal to smudge overhead code and business logic into places they don't belong before you can figure out what patterns will separate them again nicely. It's okay to be messy, but it's important to clean up afterwards with some refactoring

التعليقات

لا تمتلك صلاحيات تُمكّنك من إضافة تعليقات.