UML (Lenguaje Unificado de Modelado):

Es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad.

Es un lenguaje gráfico para **visualizar, especificar, construir y documentar un sistema**. UML ofrece un **estándar** para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos, funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y compuestos reciclados.

UML no puede compararse con la programación estructurada, pues UML significa Lenguaje Unificado de Modelado, no es programación, solo se diagrama la realidad de una utilización en un requerimiento.

No se utiliza UML solo para lenguajes orientados a objetos

En el pasado empezó como *Object-Modeling Technique* (OMT) de Rumbaugh, que era mejor para **análisis orientado a objetos**, y el Método Booch (de Grady Booch) que era mejor para el **diseño orientado a objetos**.

Los ***Tres Amigos*** **(Rumbaugh, Jacobson y Booch)** fue organizado un consorcio internacional llamado *UML Partners* en 1996 para completar las especificaciones del UML.

Tipos de diagrama UML:

Existen dos clases principales de tipos de diagramas: **diagramas *estructurales* y diagramas de *comportamiento*.**

Estructurales

Los diagramas estructurales **muestran la estructura estática del sistema y sus partes en diferentes niveles de abstracción.** Existen un total de siete tipos de diagramas de estructura:

Diagrama de clases

Los diagramas de clase son, sin duda, el tipo de diagrama UML más utilizado.

Muestra las clases en un sistema, atributos y operaciones de cada clase y la relación entre cada clase. Una clase tiene tres partes, nombre en la parte superior, atributos en el centro y operaciones o métodos en la parte inferior. **Las diferentes relaciones entre las clases se muestran por diferentes tipos de flechas.**

Diagrama de componentes

Muestra la **relación estructural de los componentes de un sistema de software**. Los componentes se comunican entre sí mediante **interfaces**. Las interfaces se enlazan mediante **conectores**.

Diagrama de despliegue

Muestra el **hardware de su sistema y el software de ese hardware**. Los diagramas de implementación son útiles cuando la solución de software se despliega en varios equipos, cada uno con una configuración particular y única.

Diagrama de objetos

Muestran la **relación entre los objetos, pero usan ejemplos del mundo real**. Se utilizan para **mostrar cómo se verá un sistema en un momento dado**. Debido a que hay datos disponibles en los objetos, a menudo se utilizan para explicar relaciones complejas entre objetos.

Diagrama de paquetes

Representa las dependencias entre los paquetes que componen un modelo. Es decir, muestra **cómo un sistema está dividido en agrupaciones lógicas y las dependencias entre esas agrupaciones**.

Dado que normalmente un paquete está pensado como un directorio, los diagramas de paquetes **suministran una descomposición de la jerarquía lógica** de un sistema

Diagrama de perfiles

Diagrama de perfil es un nuevo tipo de diagrama introducido en UML 2. Este es un tipo de diagrama que se utiliza muy raramente en cualquier especificación.

Diagrama de estructura compuesta

Los diagramas de estructura compuesta se utilizan para **mostrar la estructura interna de una clase**.

De comportamiento

Muestran el **comportamiento dinámico de los objetos en el sistema.**

Diagrama de actividades

Representan los flujos de trabajo de forma gráfica. Pueden utilizarse para describir el flujo de trabajo empresarial o el flujo de trabajo operativo de cualquier componente de un sistema.

Diagrama de casos de uso

Ofrecen una visión general de los actores involucrados en un sistema, las diferentes funciones que necesitan esos actores y cómo interactúan estas diferentes funciones.

Diagrama de máquina de estados

Útiles para **describir el comportamiento de los objetos que actúan de manera diferente de acuerdo con el estado en que se encuentran** en el momento.

Diagrama de interacción

Los diagramas de interacción incluyen distintos tipos de diagramas:

Diagrama de secuencia

Muestran **cómo los objetos interactúan entre sí y el orden en que se producen esas interacciones**. Es importante tener en cuenta que muestran las **interacciones para un escenario en particular**. Los **procesos** se representan **verticalmente** y las **interacciones** se muestran como **flechas**.

Diagrama de comunicación

El diagrama de comunicación se llamó diagrama de colaboración en UML 1. Es similar a los diagramas de secuencia, pero **el foco está en los mensajes pasados entre objetos**.

Diagrama de tiempos

Representan el comportamiento de los objetos en un marco de tiempo dado. Si es solo un objeto, el diagrama es **directo**, pero si hay más de un objeto involucrado, también se pueden usar para mostrar interacciones de objetos durante ese período de tiempo.

Diagrama global de interacciones

Los diagramas de interacción muestran una secuencia de diagramas de interacción. En términos simples, pueden llamarse una **colección de diagramas de interacción y el orden en que suceden**.

Ways of using UML:

There are different ways in which people want to use it, differences that carry over from other graphical modeling languages . These differences lead to long and difficult arguments about how the UML should be used . To untangle this, Steve Mellor and I independently came up with a characterization of the **three modes in which people use the UML** : **sketch**, **blueprint**, and **programming language** .The most common of the three is UML as sketch .

Using UML as a SKETCH:

In this usage, developers **use the UML to help communicate some aspects of a system** . You can use sketches in a **forward-engineering** or **reverse-engineering direction** .

Forward engineering draws a UML diagram before you write code
Your aim is to use the sketches to help communicate ideas and alternatives about what you're about to do . You **don't talk about all the code you are going to work on**, only important issues

Reverse engineering builds a UML diagram from existing code.
With reverse engineering, **you use sketches to explain how some part of a system works**. You **don't show every class**, simply those that are interesting and worth talking about before you dig into the code .

The essence of sketching is selectivity.

Sketching is pretty informal and dynamic, you need to **do it quickly and collaboratively, so a common medium is a whiteboard** . Sketches are also useful in documents, in which case the focus is communication rather than completeness .Often people aren't too particular about keeping to every strict rule of the UML.

Using UML as a BLUEPRINT:

UML as blueprint is about completeness .

Forward engineering, the idea is that **blueprints are developed by a designer whose job is to build a detailed design for a programmer to code up** .
The designer may be the same person as the programmer, but usually the designer is a more senior developer who designs for a team of programmers . Blueprinting **may be used for all details, or a designer may draw blueprints to a particular area** . A common approach is for a designer to develop blueprint level models as far as interfaces of subsystems but then let developers work out the details of implementing those details .

Forward-engineering tools support diagram drawing and back it up with a repository to hold the information .

Reverse engineering, **blueprints aim to convey detailed information about the code** either in paper documents or as an interactive graphical browser .
The blueprints can **show every detail about a class in a graphical form** that's easier for developers to understand .

Reverse-engineering tools read source code and Interpret from it into the repository and generate diagrams .

Blueprints require much more sophisticated tools than sketches do in order to handle the details required for the task .

Specialized CASE (computer-aided software engineering) tools fall into this category, although the term CASE has become a dirty word, and vendors try to avoid it now .
Tools that can do both forward and reverse engineering are referred to as **round-trip tools** .

Some tools use the source code itself as the repository and use diagrams as a graphic viewport an the code . These tools tie much **more closely into programming** and often integrate directly with programming editors, i like to think of these as **tripless tools** .

The line between blueprints and Sketches is somewhat blurry, but the distinction, i think, rests on the fact that **sketches are deliberately incomplete, highlighting important information**, while **blueprints intend to be comprehensive, often with the aim of reducing programming to a simple and fairly mechanical activity**. In a sound bite, I'd say that *sketches are explorative, while blueprints are definitive*

Using UML as a PROGRAMMING LANGUAGE:

Many CASE tools do some form of code generation, which automates building a significant part of a . system . Eventually, however, you reach **the point at which all the System can be specified in the UML, and you reach UML as programming language** .

Developers draw UML diagrams that are compiled directly to executable code, and the UML becomes the source code .

Model Driven Architecture (MDA) and Executable UML:

MDA:

MDA is a Standard Approach to using the UML as a programming language ; the standard is controlled bv the OMG, as is the UML . By producing a modeling environment that conforms to the MDA, **vendors can create models that can also work with other MDA-compliant environments** .

You dont have to use MDA to use UML.

MDA divides development work into two main areas:

- Modelers represent a particular application by creating a **Platform Independent Model (PIM)** .
- The **PIN is a UML model that is independent of anv particular technology**. Tools can then **turn a PIM into a Platform Specific Model (PSM).**

If you want to create a warehousing system using MDA you need to create a **single PIM** and then **multiple PSM, one for each platform**.

If the process is not automated then its blueprint and not uml as a programming language.

Executable UML:

Executable UML is similar to MDA but uses slightly different terms. Similarly, you **begin with a platform-independent model that is equivalent to MDA's PIM** .
However, the next step is to use a **Model Compiler** to **turn that UML model into a deployable system in a single step** ; hence, there's no need for the PSM .

The **model compilers are based on reusable archetypes** . An **archetype** describes how to take an executable UML model and turn it into a particular programming platform .

You need 1 archetype per platform.

It's worth using the UML as a programming language only if it results in something that's significantly more productive than using another programming language .

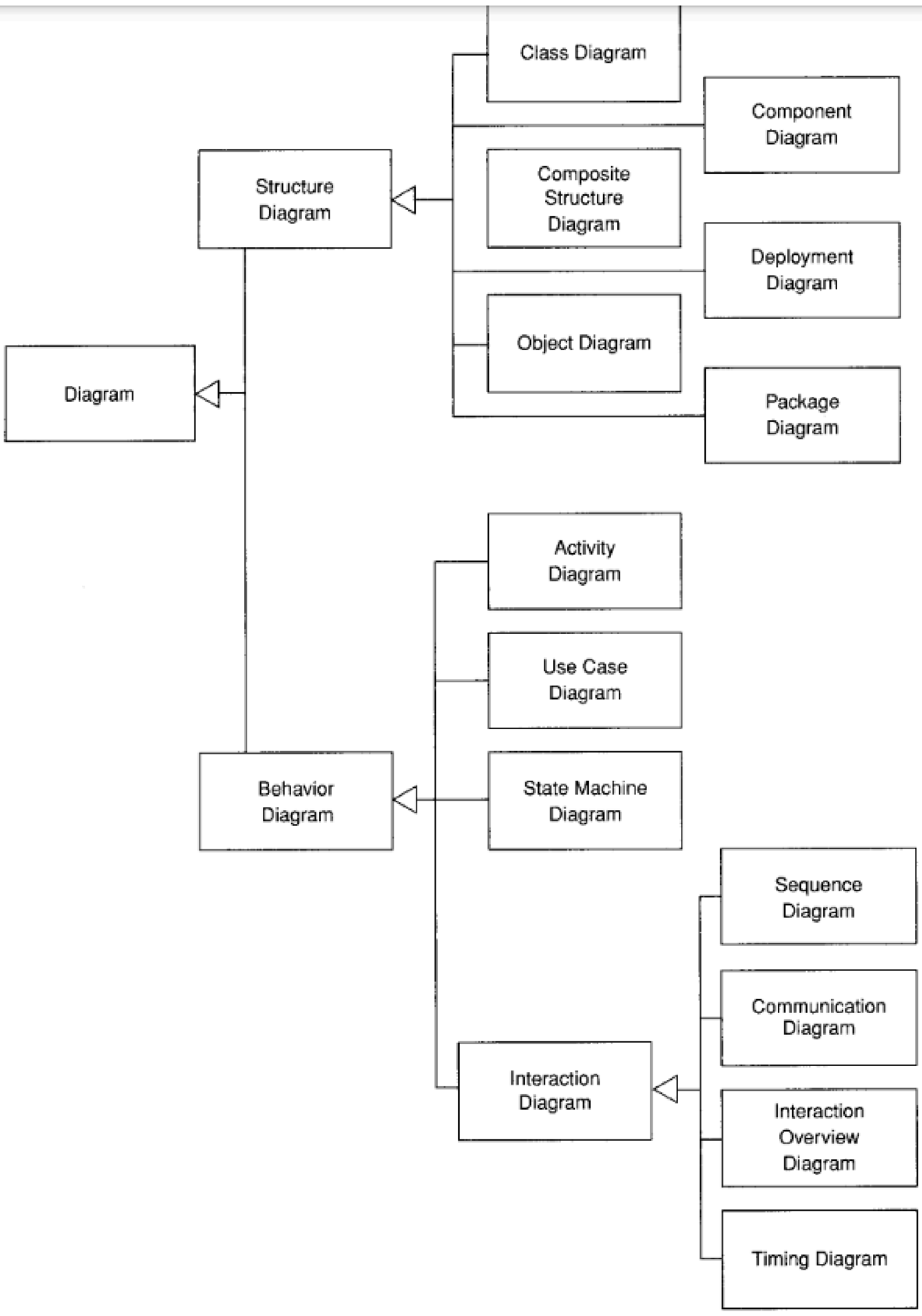
Software perspective, the elements of the UML map pretty directly to elements in a software system .

Conceptual perspective, the UML represents a description of the concepts of a domain of study. Here, we aren't talking about software elements so much as we are building a vocabulary to talk about a particular domain .

UML Diagrams

Table 1.1 *Official Diagram Types of the UML*

Diagram	Book Chapters	Purpose	Lineage
Activity	11	Procedural and parallel behavior	In UML 1
Class	3, 5	Class, features, and relationships	In UML 1
Communication	12	Interaction between objects; emphasis on links	UML 1 collaboration diagram
Component	14	Structure and connections of components	In UML 1
Composite structure	13	Runtime decomposition of a class	New to UML 2
Deployment	8	Deployment of artifacts to nodes	In UML 1
Interaction overview	16	Mix of sequence and activity diagram	New to UML 2
Object	6	Example configurations of instances	Unofficially in UML 1
Package	7	Compile-time hierarchic structure	Unofficially in UML 1
Sequence	4	Interaction between objects; emphasis on sequence	In UML 1
State machine	10	How events change an object over its life	In UML 1
Timing	17	Interaction between objects; emphasis on timing	New to UML 2
Use case	9	How users interact with a system	In UML 1


Class Diagram
Component Diagram
Structure Diagram
Composite Structure Diagram
Deployment Diagram
Diagram
Object Diagram
Package Diagram
Activity Diagram
Use Case Diagram
Behavior Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Diagram
Interaction Overview Diagram
Timing Diagram

Development Process:

When you hear people discuss the UML, you often hear them talk about the **Rational Unified Process (RUP)** . RUP is one process-or, more strictly, a process framework-that you can use with the UML. But other than the common involvement of various people from Rational and the name "unified," it doesn't have any special relationship to the UML . **The UML can be used with any process** . RUP is a popular approach.

Iterative and Waterfall Processes

The essential **difference between the two is how you break up a project into smaller chunks** .

The **waterfall style** breaks down a project based on an activity. To build Software, you have to do certain activities : **requirements analysis, design, coding, and testing** . *A project can breakdown those activities on multiple phases*.

The **iterative style** breaks down a project by subsets of functionality . In the first iteration, you'd **take a quarter of the requirements and do the complete software life cycle for that quarter : analysis, design, code, and test** . At the end of the first iteration, you'd have a system that does a quarter of the needed functionality . Then you'd do a second iteration so that at the end of 6 months, you'd have a system that does half the functionality .

On all each iteration you do a part of the requirements and all the activities needed for it, then in the next one you will do the same and so on till you finish the project.

It's inevitable that analysis and design decisions will have to be revisited in later phases, these backflows are exceptions and should be minimized as much as possible .

With iteration, you usually see some **form of exploration activity before the true iterations begin** . At the very least, this will get **a high-level view of the requirements**

In iteration you may have **multiple releases, each of which is broken down into several iterations** .

You can have **hybrid approaches** . It describes the staged delivery life cycle whereby **analysis and high-level design are done first, in a waterfall style**, and then the **coding and testing are divided up into iterations** .

It's very difficult to tell whether the project is truly an track with a waterfall process

Most oftenly **ITERATIVE > WATERFALL**

It is particularly important that each iteration produces tested, integrated code that is as dose to production quality as possible .

One of the most common concerns about iterative development is the issue of **rework.**

Predictive and Adaptive Planning

A **predictive approach** looks to do work early in the project in order to yield a greater **understanding of what has to be done later** . This way, you can reach a point where the latter part of the project can be estimated with a reasonable degree of accuracy.

With **predictive planning**, a project has two stages . The **first stage comes up with plans and is difficult to predict**, but the **second stage is much more predictable because the plans are in place** .

As the project goes on, you gradually get more predictability.

And even once you have a predictive plan, things will go wrong . You simply **expect that the deviations become less significant** once a solid plan is in place .

One of the unique sources of complexity in software projects is the **difficulty in understanding the requirements for a software system** .

The majority of software projects experience significant **requirements churn** : **changes in requirements in the later stages of the project** . These changes shatter the foundations of a predictive plan . You can **combat these changes by freezing the requirements early and not permitting changes, but this runs the risk of delivering a system that no longer meets the needs of its users** .

Adaptive planning, whereby **predictivity is seen as an illusion** . Instead of fooling ourselves with illusory predictability, we should **face the reality of constant change and use a planning approach that treats change as a constant in a software project** .

This **change is controlled so that the project delivers the best software it can** ; but although **the project is controllable, but it is not predictable** .

With a predictive plan, you can develop a fixed-price/fixed-scope contract .
Such fixing isn't possible with an adaptive plan .

Two important pieces of advice:

1 . **Don't make a predictive plan until you have precise and accurate requirements** and are confident that they **won't significantly change** .
2 . **If you can't get precise, accurate, and stable requirements, use an adaptive planning** style .

Agile Processes

Examples of these processes are Extreme Programming (XP), Scrum, Feature Driven Development (FDD), Crystal, and DSDM (Dynamic Systems Development Method) .

Agile processes are **strongly adaptive in their nature**. They are also very much **people-oriented processes** . Agile approaches assume that the most important factor in a project's success is **the quality of the people on the project** and how well they work together in human terms . Which process they use and which tools they use are strictly second-order effects .

Agile methods tend to use **short, time-boxed iterations**, most often of a month or less . Because they don't attach much weight to documents, **agile approaches disdain using the UML in blueprint mode** . **Most use the UML in sketch mode**, with a few advocating using it as a programming language .

Agile processes tend to be **low in ceremony**. **Few documents and control points during the project** . Agile processes consider that ceremony makes it harder to make changes and works against the grain of talented people .It's important to realize that **the lack of ceremony is a consequence of adaptivity and people orientation** rather than a fundamental property.

Rational Unified Process

Although **RUP is a process framework**, **providing a vocabulary and loose structure to talk about processes** .

When you use RUP, the first thing you need to do is **choose a development case** : the process you are going to use in the project. **Development cases can vary widely**.

RUP is essentially an iterative process . A **waterfall style isn't compatible with the philosophy of RUP**.

Four Phases of RUP Projects:

1 . **Inception** makes an initial evaluation of a project . Typically in inception, you decide whether to commit enough funds to do an elaboration phase .
2 . **Elaboration** identifies the **primary use cases of the project** and **builds software in iterations** in order to shake out the architecture of the system .
3 . **Construction** continues the building process, **developing enough functionality to release** .
4. **Transition** includes various **late-stage activities that you don't do iteratively**.
These may include deployment into the data center, user training, and the like

Fitting a Process to a Project

Consequently, you always have to **adapt a process to fit your particular environment** .

One of the first things **you need to do is look at your project and consider which processes seem close to a fit** . This should give you a short list of processes to consider.

You should then **consider what adaptations you need to make to fit them to your project**.

Then you can **start modifying the process** . If from the beginning you are more familiar with how a process works, you can modify it from the beginning .

However confident you are with your process when you begin, **it's essential to learn as you go along**.

Indeed, one of the great benefits of **iterative development is that it supports frequent process improvement** .

At the **end of each iteration, conduct an iteration retrospective**, whereby the team assembles to consider how things went and how they can be improved .

A good way to do this is to make a **list with three categories** :
1 . **Keep**: things that worked well that you want to ensure you continue to do
2 . **Problems** : areas that aren't working well
3 . **Try**: changes to your process to improve it

Fitting the UML into a Process

Using the UML doesn't necessarily imply developing documents or feeding a complex CASE tool . Many people draw UML diagrams on whiteboards only during a meeting to help communicate their ideas .

Requirements Analysis

The activity of requirements analysis **involves trying to figure out what the users and customers of a software effort want the System to do** .

A number of UML techniques can come in handy here :
• **Use cases**, which describe how people interact with the System .
• A **class diagram** drawn from the conceptual perspective, which can be a good way of building up a rigorous vocabulary of the domain .
• An **activity diagram**, which can Show the work flow of the organization, showing how Software and human activities interact . An activity diagram can Show the context for use cases and also the details of how a complicated use case works .

• A **state diagram**, which can be useful if a concept has an interesting life cycle, with various states and events that change that state .

Design

When you are doing design, you can get more technical with your diagrams . You can use more notation and be more precise about your notation .

Some useful techniques are:
• **Class diagrams** from a software perspective . These show the classes in the software and how they interrelate .
• **Sequence diagrams** for common scenarios . A valuable approach is to pick the most important and interesting scenarios from the use cases and use CRC cards or sequence diagrams to figure out what happens in the software .
• **Package diagrams** to show the large-scale organization of the software .
• **State diagrams** for classes with complex life histories .
• **Deployment diagrams** to show the physical layout of the software .

With a waterfall life cycle, you would **do these diagrams and activities as part of the phases** . The end-of-phase documents usually include the appropriate UML diagrams for that activity . A **waterfall style usually implies that the UML is used as a blueprint** .

In an iterative style, the UML diagrams can be used in either a blueprint or a sketch style . With a blueprint, the analysis diagrams will usually be built in the iteration prior to the one that builds the functionality . Each iteration doesn't start from scratch ; rather, it modifies the existing body of documents, highlighting the changes in the new iteration .

Blueprint designs are usually done early in the iteration and may be done in pieces for different bits of functionality that are targeted for the iteration .

Using the UML in sketch mode implies a more fluid process. One approach is to spend a couple of days at the beginning of an iteration, sketching out the design for that iteration . You can also do short design sessions at any point during the iteration, setting up a quick meeting for half an hour whenever a developer starts to tackle a nontrivial function .

Documentation

Once you have built the Software, **you can use the UML to help document what you have done**. For this, i find UML diagrams useful for getting an overall understanding of a system . In doing this, however, i should stress that i do **not believe in producing detailed diagrams of the whole system** .

You should write additional documentation to highlight important concepts . For detailed use documentation generated from code.

A **package diagram** makes a good logical road map of the system . This diagram helps nie understand the logical pieces of the system and see the dependencies and keep them under control .
A **deployment diagram**, which shows the high-level physical picture, may also prove useful at this stage .

Within each package, i like to see a class diagram . I show only the important features that help me understand what is in there .

The **class diagram should be supported by a handful of interaction diagrams** that show the most important interactions in the system .

Understanding Legacy Code

The UML can help you figure out a gnarly bunch of unfamiliar code in a couple of ways .

Building a **sketch of key facts** can act as a graphical note-taking mechanism that helps you capture important information as you learn about it . **Sketches of key classes** in a package and their key interactions can help clarify what's going on .

A particularly nice capability is that of generating a sequence diagram to see how multiple objects collaborate in handling a complex method .

Choosing a Development Process

I'm strongly in favor of iterative development processes . As I've said in this book before : You should **use iterative development only an projects that you want to succeed** .

Done well, it is an essential technique, one **you can use to expose risk early and to obtain better control over development** . It is not the same as having no management, although to be fair, i should point out that some have used it that way. It does need to be well planned . But it is a solid approach, and every 00 development book encourages using it-for good reason. You should not be surprised to hear that as one the authors of the Manifesto for Agile Software Development, I'm very much a fan of agile approaches . I've also had a lot of positive experiences with Extreme Programming, and certainly you should consider its practices very seriously.