Table of Contents for Chapter 7

<<< … >>>

Part 2. Architect's Toolbox - 'bird's view of the terrain'

Imagine scenario – you are a CIO or the Project Manager, and you've got a couple of techos from your team in the boardroom for a vital technical discussion. You are responsible for the delivery to business of the large mission-critical application on time, on budget and to the business satisfaction. Your seat is on the line and you have to make some hard and fast technical decisions. You are not really an expert IT Architect or Software Engineer, and do not have to be.
Your techos locked in a technical argument, spitting unknown to you TLAs (Three Letter Acronyms), and even FLAs, at will. You are loosing the track of discussion. You resort to carefully watching their facial expressions and struggling to maintain a meaningful expression of your own. Would not it be great to catch the TLA and throw it right back at them?

Or, there is another scenario. You are the IT Solution Architect responsible for putting together the large and complex application. This application cuts across various technologies and products. You know your stuff and you've got a track record to prove it. It is just that there are some technologies and products that you did not come across before.
No problem whatsoever. You have to pick it up quickly, as you've done it many times before throughout your career. Let me just find a concise and to the point reference for you.

Or, there is yet another scenario for you. You are the student, or the fresh starter in this industry or in particular technology pocket. You need a practical guide to get you going quickly.

IT Architect needs to be aware of multitude of technologies, methodologies and products. And this field is always a moving target. You need years and years of hard learning and sweaty experience (and at times even bloody experience). You cannot acquire such valuable knowledge and skills for nothing, without doing it tough.

You did not really think that someone could give it to you on a plate, or in a few hundred of pages of a reference book, did you? It would be preposterous to think that all major technologies may be covered in a brief summary. Don't be ridiculous and get over it… What? …Well, OK. As you wish... Only because you insist…

In this part, we peek into the IT Architect's toolbox and make a stock take of his or her tools of trade – methodologies and techniques, standards and protocols, platforms and technologies. This is a 10,000 feet helicopter view to help you with the big picture.

Chapter 7. Methodologies and Techniques

This chapter takes a broad swipe at concepts and approaches that get you organised in a search of the best solution.
We need common guidelines for devising and presenting our designs. Also, we need to get the team organised and arm them with common notation and semantics of design best practices. This way we can produce, capture and document our designs, and communicate it to our peers.
In other words, we concentrate in this chapter on methodologies, techniques and major patterns that are applicable to the Enterprise Architect.

Chapter explains fundamentals of some approaches and methodologies that (in author's mind at any rate) captured significant market and mind share of software developers and integrators. This is not a complete stock take of the software methodology landscape.
For example, chapter does not cover such important approaches as OPEN Process, Object-Oriented Software Process (OOSP, do not confuse with the object-orientation in general – we do explain that), Catalysis (Desmond D'Souza), Dynamic System Development Method (DSDM), Crystal Methodologies (Alistair Cockburn), SCRUM (Jeff Sutherland), Adaptive Software Development (Jim Highsmith), Feature-Driven Development (FDD, Jeff De Luca and Peter Coad), Enterprise Unified Process (EUP), Navigator.

Modular and Structured – 'ancient history?'

Methodologies and techniques in the Art of the Software Creation have undergone rapid evolution. Software practitioners, who joined our vibrant IT community this millennium, sometimes under-appreciate (to say the least) the challenges and achievements of those who did IT before us and brought us where we are now.
However, there is knowledge to absorb and there are lessons to learn – for the better understanding of modern challenges in IT.

First programmers and software designers were so overwhelmed by the sheer excitement of making computer do the work in the first place, that IT quickly reached the point where convoluted code and unruly software discredited IT all too often.
Business demanded to make the software development process and its deliverables manageable, disciplined, rigorous and of predictable high quality.

With IT penetrating the vital fabric of business and society, program bugs and programmers' mischief was not a laughing matter anymore (well, at least not every time).

With every new project, programmers created universes to their liking. Often only the creator could understand the program code.
With complex logic, control flow, mysterious variables and web of branches, program code reminded 'spaghetti'. If program worked somehow, nerd was forgiven and grudgingly tolerated, feared and envied.
IT industry was heading for its first crisis and required a good soul-searching to win the trust of the business. Sounds farfetched?

Consider this loop statement in Fortran:

```
DO 18 I=1,4
```

Imagine likely situation when programmer (or punch card typist who does not know Fortran and reads from the programmer's scribbles) omitted comma or typed 'dot' instead:

```
DO 18 I=1.4
```

'Luckily', Fortran compiler does not have reserved keywords, and ignores spaces as well. If variable was not explicitly declared, first mention of the new variable is considered to be its declaration. So, we get an innocent assignment statement here instead of the loop header:

```
DO18I=1.4
```

You say, tough luck? We say, this error in the Fortran program of the flight control software caused disaster in the first launch of the US spacecraft to Venus [Mayers 1976].

Computer scientists, mathematicians and software methodologists embarked to introduce rigour in programming practices and software design.

Ideas of *Structured Programming* were succinctly formulated by Dijkstra in [Dijkstra 1968a].
However, foundation for the *Structured Programming* was built in earlier works by Bohm and Jacopini [Bohm 1965] – they proved the *theorem of structuring*.
This *theorem* established that logical structure of any complexity could be build from the small number of basic elementary structures of the control flow. And these basic logical structures were actually identified.

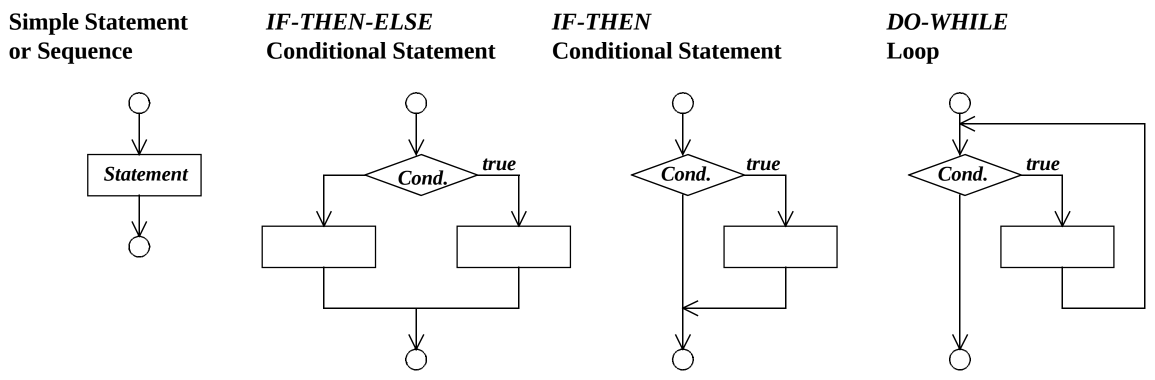

Simple Statement or Sequence
Statement
IF-THEN-ELSE Conditional Statement
Cond.
true
IF-THEN Conditional Statement
Cond.
true
DO-WHILE Loop
Cond.
true

Figure 7.1. Basic Logical Structures of the Program

Note that each basic structure has one entry point and one exit point.
Statement may be a simple assignment, subroutine or function call, input-output statement or any other basic logical structure. This way, through the nesting and recursion, any complex program logic may be constructed.

Minimal, theoretically sufficient set of basic logical structures imposed unnecessary constraints on the programmer's freedom of expression.
Some pragmatic additional elementary logical building blocks were introduced.

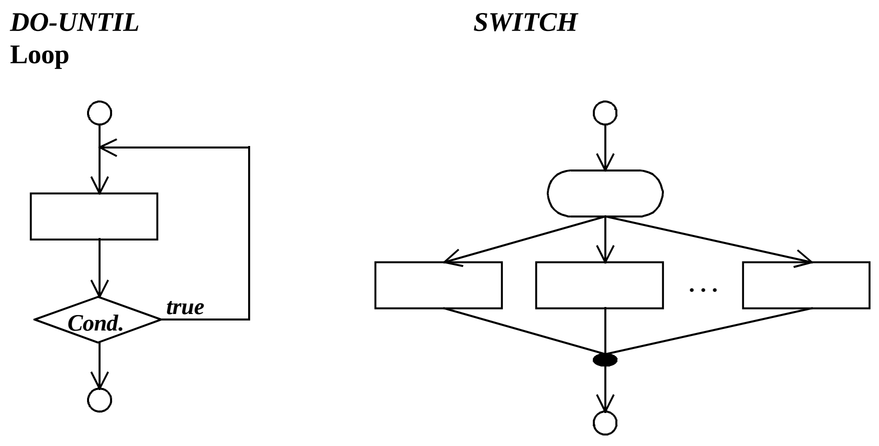


Figure 7.2. Additional Logical Structures of the Program

As a side effect of bringing the *Structured Programming* into mainstream software development practices, lively debacle on the role and evils of the *GOTO* statement came about.

GOTO statement caused program to skip the 'normal' control flow and for address pointer to jump directly into some nominated place in the program (usually explicitly denoted by the label assigned to some other statement).
Nothing prevented you from jumping out of the loop, inside the loop or inside the conditional clause.
GOTO statement was singled out as a major culprit in producing the incomprehensible 'spaghetti code'. Drawbacks and benefits of the *GOTO* statement were the topic of heated debates, eg. see [Dijkstra 1968b].
New programming languages trumpeted absence of the *GOTO* statement as a major marketing point.
Those that still allowed *GOTO* - tried to bury this feature deep in the documentation, in shame.

If you do not have to think much about the *Structured Programming* while writing your code now – this is because modern programming languages to the great extent *enforce* the *Structured*

Programming for you, and *force* you to do the right thing. *Structured Programming* features are built-in into the modern programming languages.
This is not to say that modern programmer cannot find a way of writing the convoluted and poorly structured code. There is no foolproof system that is completely immune from the fool (or, if you like, the one, way too clever for his or her own good). However, good development environment helps us to manage the situation better.

Structured Programming deals with the monolithic chunk of the program source code – in the same address space or the same process, with all variables visible throughout the single program in question.
Obviously, *Structured Programming* is limited in scope to every single chunk of the source code.
Project rarely consists of one single program, rather of multitude of collaborating programs or *modules*.
This is where *Modular Programming* techniques close the gap.

Modular Programming provides guidelines for designing your software as a set of manageable *modules*.
Modules are (usually and preferably) small, about 50 lines of code, programs that implement some clearly defined business function. Bigger function can be implemented by calling several smaller functions in order – as per *Structured Programming*.
Modules are implemented as sub-routines with interfaces and rules of how the module can be invoked.

Still, a lot was left to the programmer's discretion and his or her diligence in following the wise guidelines of the program design.
For example – program designer had to ensure that the *module* communicates with the calling routine through the input and output parameters in the *CALL* statement for this *module*. Other options for the *module* invocation could be – passing parameters through some persistent stored values, or visible global variables – a big no-no as far as *Modular Programming* concerned.
If the *CALL*-interface is simple and stable, we can easily test, improve or replace the *module*, without the adverse impact on its calling environment.
Modules may be packaged and shipped as a library of programs integrated into many different projects.

Does this sound to you like *insulation*, *encapsulation*, *separation of concerns* and *re-use*? It should, because *Modular Programming* is exactly that.
Despite what you may have heard or read, *Object-Oriented Methodology* (with the due respect and recognition of all the innovations that yet to be mentioned here) is less of the revolution and more of the evolution for those who went through the motions of *Modular Programming*.

Object-Oriented Methodology

Computing abstractions progressed through several distinct stages:

- Bare hardware with the binary set of instructions that controls it. Assembler languages introduced human-readable mnemonic to binary operations and data – strictly to keep human (the weakest link in the computerised world) happy
- Low-level programming that is just a little bit more human-friendly – second generation programming languages and methodologies
- Higher level programming notation that allows us to describe complex algorithms on some elementary and aggregated data types – still far cry from the level of abstraction in the real-life processes

Next natural stage in the evolution of computing notation inevitably delivered paradigm of higher level of abstraction, closer to the human mindset and human natural way of thinking.
This progression became possible due to the advancements in computer technology and growing human appetites for the more natural toolset, without levels of indirection using somewhat alien to human mind information models.

It came as no surprise, as we always wanted software to evolve towards the higher level of abstraction, closer to the real-world entities. We always wanted software artefacts to resemble the real life closer, so that we could model, monitor and control real-life processes by manipulating entities and processes that are better understood.

Removal of levels of indirection in the software paradigm will reduce complexity in the software system.

Object-Oriented paradigm is a new way of thinking when we view software system as a universe of interacting autonomous agents.
Every agent performs some tasks, communicates with other agents through the set of messages and may possess some memory about itself, other agents, about communications with other agents and the surrounding environment.
Collaborations of agents deliver the functionality required of the software system.

This agent is *The Anthropomorphic* (or human-like) *Object* – fundamental notion and the starting point in the Object-Oriented Methodology.

Object represents concept or entity from the real world. *Object* is the unit of encapsulation for both *state* and *behaviour*.

Each *Object* is a unique *instance* and has its own distinct *identity*.
State of the *Object* represented by properties (attributes) and their values.

Behaviour of the *Object* is defined as a set of *methods* (procedures, operations, messages) that perform its responsibilities.
External behaviour of the *Object* is defined by a set of messages that *Object* exposes to the outside world – known as Object's *public interface* (or contract, protocol, specification). Objects exchange *messages* by invoking *methods* on each other.

Notion of *type* in conventional 3GL programming languages has undergone a major evolution in the Object-Oriented paradigm.
Class is a higher abstraction that defines a type, or shape if you like, or public interface, or state and behaviour of the *Object* instance.

Main source of confusion for the novice in the object-orientation is in the losing track of what level of abstraction the current model is on – *Class* or the *Object* instance.
Object is the entity itself, or model. *Class* is the information about the type of possibly many different *Objects*, or meta-model (*meta* means data about data).

Make sure you always know which level of abstraction you are on (*Class* or *Object*) – and you can't go too wrong. Accordingly, any diagram of your object-oriented design will be either *Object* or *Class* diagram – have both, but do not mix two.
Furthermore, by level of abstraction we often mean level of detail, fidelity or granularity of our models. We distinguish coarse-granular and fine-granular models – with latter being an extension, drill-down elaboration of the former. Do not mix different levels of detail on your diagrams - 'divide and conquer'.

As you can see by now, Object-Orientation is a complete philosophy or a whole new vision of the world. Out of vast literature on object-orientation, some of the most often referred titles are [Booch 1994] [Meyer 1997].
Object-Oriented Methodology leaves the object-oriented designer with a 'minor' task (we wish) of identifying types of entities (*Classes*) and instances (*Objects*) from the real world, their behaviours and collaborations between them.

Analysis of the information domain and identifying *Classes* in your object-oriented model is the creative human activity that cannot be automated.
Object-oriented methodologies suggest the use of CRC (Class, Responsibilities, Collaborators) Cards to facilitate the process of initial analysis. CRC Cards were first proposed in RDD (Responsibility-Driven Design, Rebecca Wirfs-Brock).
In a brainstorming session, team fills one CRC Card for each potential *Class*. CRC Card defines the class name, its behaviours or responsibilities and its relationships with other classes.

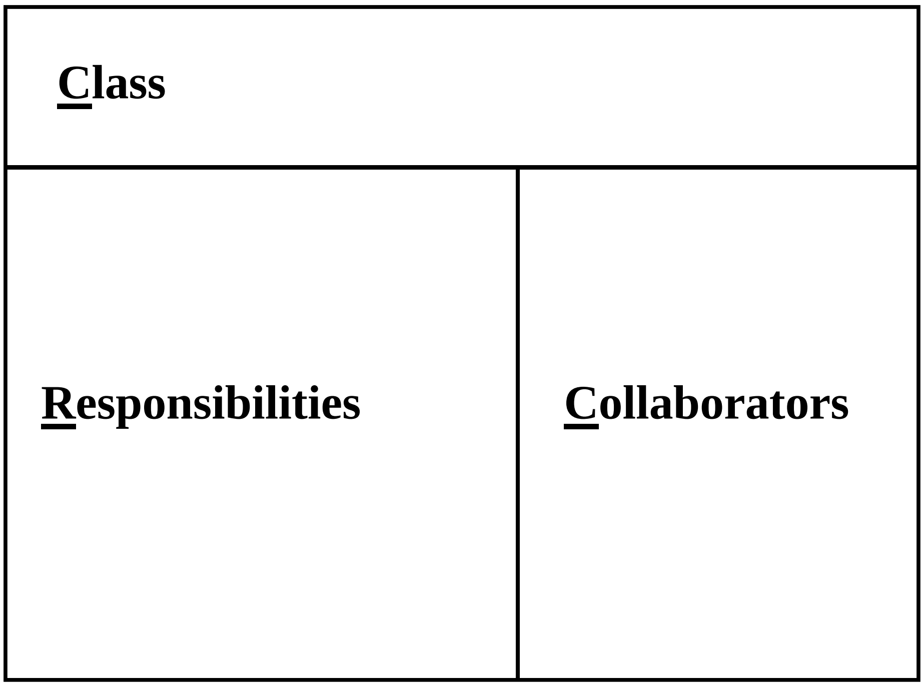


Figure 7.3. CRC Card Layout

If we look with the magnifying glass at the collaborations between *Objects* or between *Classes*, we find that any collaboration is the Client-Server relationship.
Every *Object* in collaboration is either Client or Server. Object may play role of the Client in some collaborations, and the Server in other.
When an *Object* requests a task to be performed by another *Object* (sends the message, invokes method on that *Object*), the requesting *Object* is known as the Client, and the performing *Object* is known as the Server.

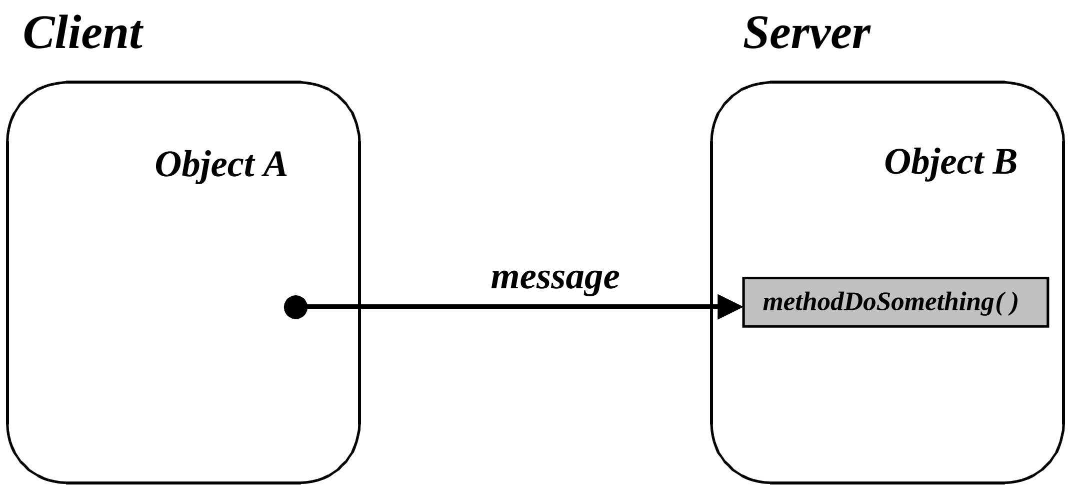


Figure 7.4. Client-Server relationships between Objects

Relationships between *Objects* and between *Classes* in the Object-Oriented Methodology may have different *semantics* (meaning).

Main types of relationships that need to be understood in order to capture the essence of object-orientation are:

- *Is kind of* or *is-a*, that denotes relationship of *Generalisation*, or *Specialisation*, or *Inheritance*
- *Is part of* or *has-a*, that denotes relationship of *Aggregation*, or *Containment*, or *Composition*

Deficiencies in the object-oriented design in most cases can be traced back to the confusion between 'kind of' type relationships (generalisation, specialisation, inheritance) and 'part of' containment relationships (aggregation, composition), and consistency in level of detail in your model.

Three pillars or major innovations and differentiating features of the Object-Oriented Methodology are *Inheritance*, *Encapsulation* and *Polymorphism*.

In *Inheritance* relationship, a sub-class 'inherits' from the super-class. A sub-class inherits protocol, methods and instance variables from its super-class. A sub-class can override inherited features of the public interface, and can provide additional features.
Inheritance relationships may create an inheritance hierarchy of related classes.

Encapsulation serves to separate a public (contractual) interface of some abstraction and its implementation. This way we build complex architectures from components by connecting them through their public interfaces. We can replace or improve the implementation of component without an impact on the rest of architecture.

Polymorphism is the object-oriented concept borrowed from the type theory wherein a programming variable can store values of different types or function can operate over different signatures.
Polymorphism is closely related to the late binding when exact type or method not determined until execution.
In fact, there is good litmus test for *Polymorphism*: the existence in the object-oriented language of a switch statement that selects an action based upon the type of the object is often a warning sign that the developer has failed to apply polymorphic behaviour effectively [Booch 1994].

Unified Modelling Language (UML)

UML History

Object-oriented modelling languages began to appear in mid-1970s as various methodologists experimented with different approaches to object-oriented analysis and design. Several other techniques influenced these languages, including Entity-Relationship data modelling (Ted Codd et al) and the Specification and Description Language (SDL, circa 1976, CCITT).
The number of identified modelling languages increased to more than 50 by 1994, fuelling 'method wars' and user dissatisfaction with any particular object method. Market matured for convergence of methods and complementing of methods with each other strengths.

The development of UML started in October 1994 when Grady Booch and Jim Rumbaugh joined forces at Rational Software Corporation and began their work on unifying the Booch and OMT (Object Modelling Technique) methods.
Despite notable differences in notation (eg, Booch method represented classes in the shape of clouds, against tastes and better judgement of many in the developer community), semantics and scope of both methods were strikingly similar.
A draft version 0.8 of the Unified Method, as it was then called, was released in October of 1995 [OMG UML 2001].
Also in 1995, Ivar Jacobson and his Objectory Company joined Rational and this unification effort with OOSE (Object-Oriented Software Engineering) method. OOSE contributed in what is known now as RUP (Rational Unified Process).

See Figure 7.5 for the genesis of UML. Figure emphasises that there are more influences to UML that we need to acknowledge, in addition to major contributions from Booch, OMT and OOSE methods.

Some other notable contributions (either initially or at the later stage and mentioned here in no particular order) were provided by such influences as RDD (Responsibility-Driven Design, Rebecca Wirfs-Brock), ROOM (Real-Time Object-Oriented Method, Bran Selic), SOMA (Semantic Object Modelling Approach, Ian Graham), Shlaer-Mellor, Coad (Peter Coad and Ed Yourdon), Martin-Odell, OPEN (Brian Henderson-Sellers, Ian Graham), Fusion (Coleman), BON (Business Object Notation with Eiffel, Walden and Nerson, Bertrand Meyer), OML (Don Firesmith), Syntropy (Cook and Daniels), Catalysis (Desmond D'Souza and Alan Wills).

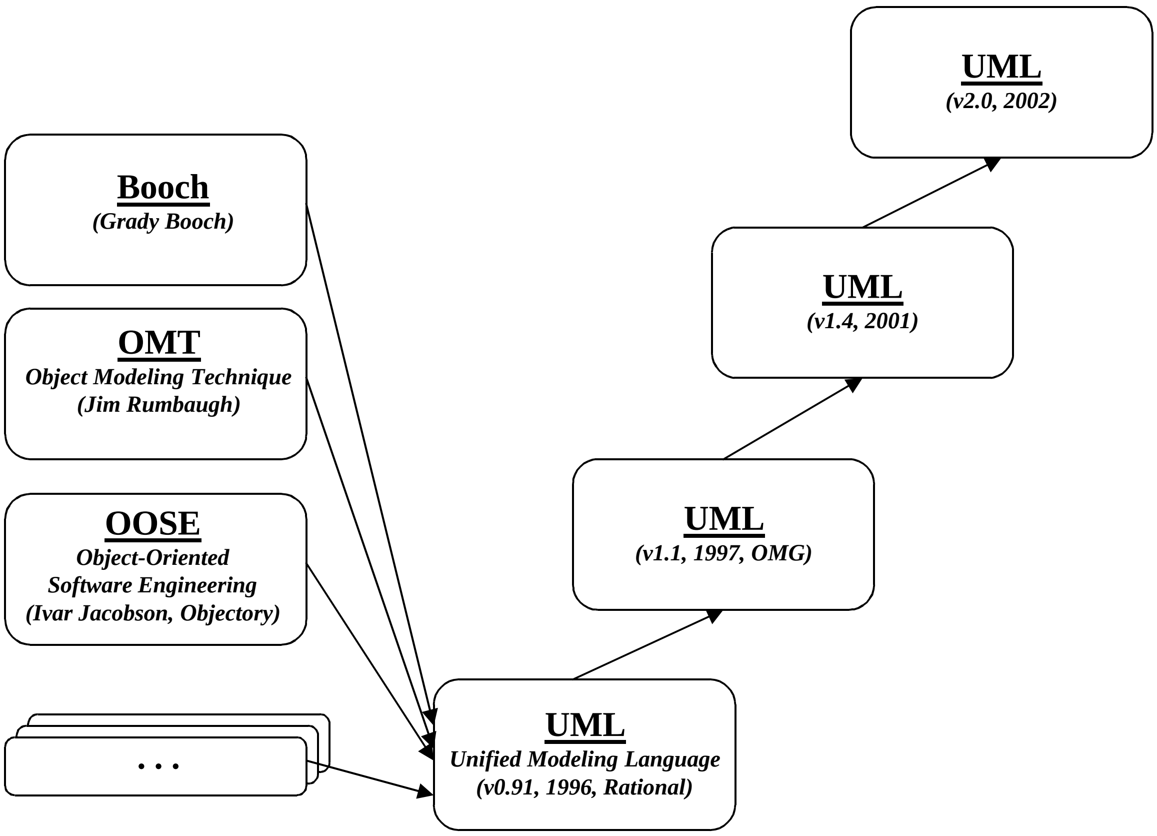


Figure 7.5. UML Evolution

In 1996, OMG issued Request For Proposal (RFP) and Rational established UML Partners Consortium with several organizations willing to dedicate resources towards a strong UML definition. Together the partners produced the revised UML v1.1 response that improved the clarity of UML and incorporated contributions from the new partners.
The UML Partners contributed a variety of expert perspectives, including, but not limited to, the following: OMG and RM-ODP technology perspectives, business modelling, constraint language, state machine semantics, types, interfaces, components, collaborations, refinement, frameworks, distribution, and metamodel.

The UML can be extended further without redefining the UML core. The UML, in its current form, is expected to be the basis for many tools [OMG UML 2001].

UML Model Elements, Notation and Semantics

OMG UML Specification [OMG UML 2001] is the official comprehensive primary source of information on UML.

Unfortunately, far from being a textbook or tutorial, UML Specification is not a bedtime reading for faint-hearted. You shall turn to [OMG UML 2001] for reference, but rely on many of more gentle and pragmatic sources like [Booch 1999].
Also, help information of UML tools (like Rational Rose, Visio, Together) will assist you in finding your way around UML.

In the end of the day, UML is just a modelling language – it does not relieve you from the necessity of grasping core concepts of object-oriented analysis and design. Tool can help you in drawing nice-looking models, but they won't worth a paper (provided you were so adventurous as to print your models), if you did not capture proper semantics of the application in the modelling language.

You start modelling of the information domain by identifying entities (together with their major behaviours and collaborations) in the real world that you will be dealing with in your application. These entities will become Model Elements.

As a language, UML defines some *Syntax* and *Semantics*.

UML Notation Guide defines *Model Elements* and how to visualise *Relationships* between them, i.e. UML *Syntax*.

UML *Semantics* (or *meaning*) defines the formal rigorous UML Language Architecture as a four-layer Metamodel Architecture [OMG UML 2001].
Four layers in the UML Language Architecture are:

- ***Meta-metamodel*** – defines the language for specifying metamodels. Aligned with OMG Meta-Object Facility (MOF)
- ***Metamodel*** – defines the language for specifying models. Operates with such artefacts as Class, Attribute, Operation and Component. Actually, in layman terms, this is the UML itself that we love and learn
- ***Model*** – defines a language to describe an information domain for our application. These are models that we design
- ***User Objects*** – defines a specific information domain. Operates with object instances

Enterprise Architects out there in trenches are usually concerned with last three layers. However, it is good to know the rigour that has been applied to the foundation of UML, and UML's potential to live up to high expectations in the near future.

It is like with any natural or very familiar activity, walking for example. We do not think about moving every muscle when putting one leg in front of another every time. Imagine caterpillar suddenly has to think about each of her 40 legs when moving. She would probably die of hunger on the spot where this thought first hit her…
So, do not get too concerned with all intricacies under the hood that were exposed to you in the UML Specification. Understanding the concepts explained here is as far as you most likely need to go.

Figure 7.6 depicts some UML Model Elements that you will use to visualise entities of your information domain in your models.

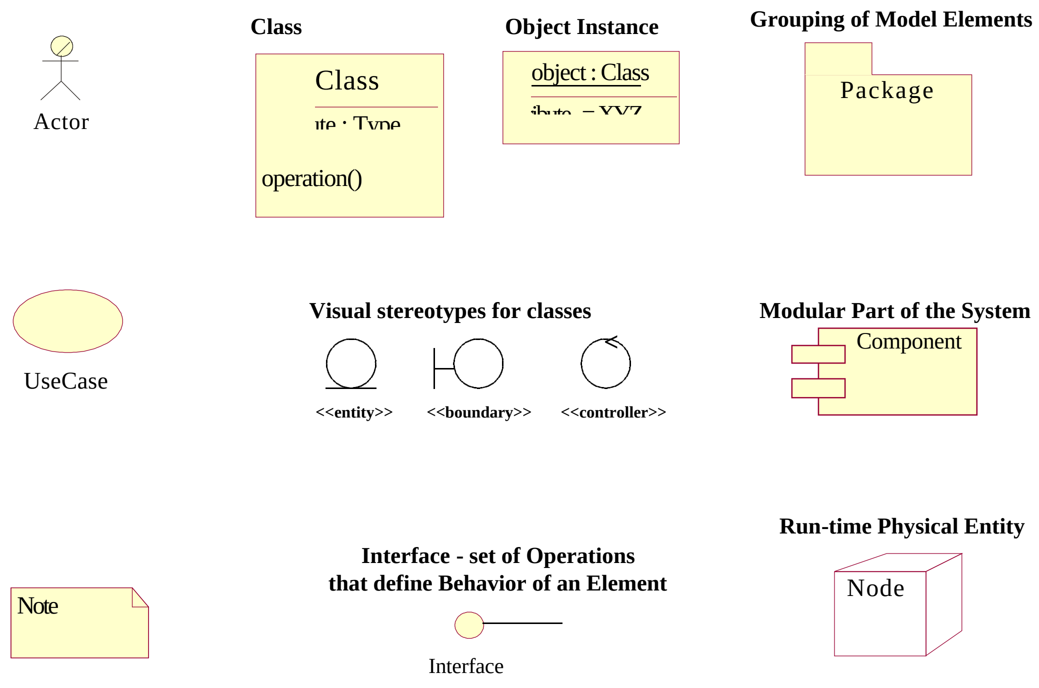
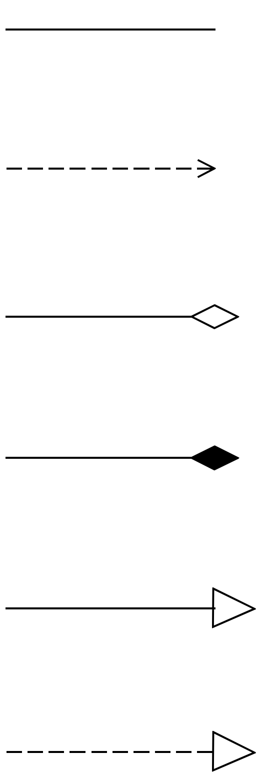


Figure 7.6. UML Model Elements

Figure 7.7 defines main types of UML Relationships between Model Elements in your model.



Association **– relationship between classes**
(may have adornments to show properties of association and its ends – name, role, cardinality/multiplicity, navigability, visibility etc.)

Dependency **– directed relationship from *client* to *supplier*, stating that *client* is dependent on *supplier***

Aggregation **- “*has-a*” or *whole*/*part* relationship**
(weak, “by-reference” – destroying *whole* does not destroy *parts*)

Composition **- strong form of Aggregation - containment**
(“by-value” – *parts* are destroyed along with the *whole*)

Generalisation **(or specialisation) –**
Inheritance “is-a” **relationship**

Realisation **- implementation of a specification**

Figure 7.7. UML Relationships

UML Relationships may have complex semantics that should be captured through many properties or adornments of Relationships.
Figure 7.8 describes some properties of UML Relationships and their notation or graphical representation. Numbers denote cardinality or multiplicity of Model Elements in Relationships.

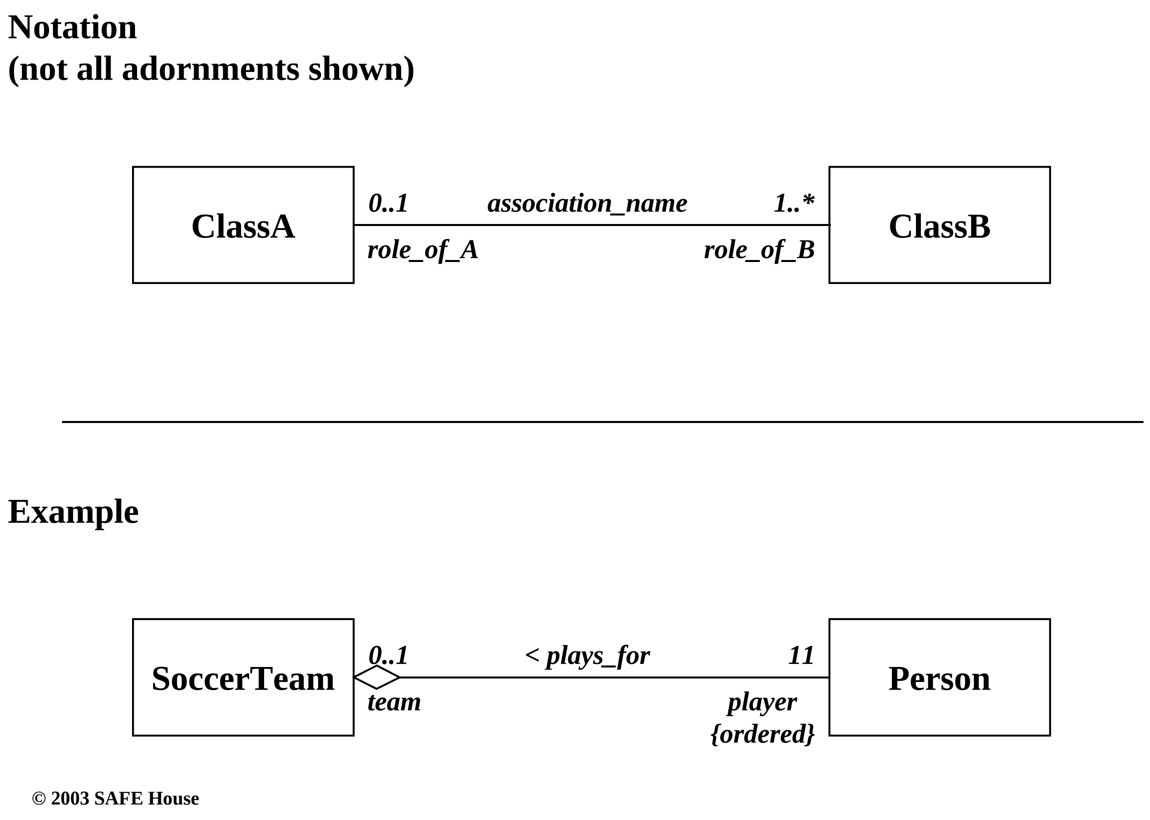


Figure 7.8. UML Association Adornments

UML Diagrams

The choice of what models and diagrams one creates has a profound influence upon how a problem is attacked and how a corresponding solution is shaped…Because of this [OMG UML 2001]:

- Every complex system is best approached through a small set of nearly independent views of a model. No single view is sufficient
- Every model may be expressed at different levels of fidelity
- The best models are connected to reality

The last point here is to remind us what we are doing in first place, just in case we forgot. Other points capture the very essence of the Software Architecture models.

UML canonises list of recommended model views or diagrams that modeller uses.
Developers often use different diagrams or name them differently. Apart from being powerful analysis and design tool, introduction of the unified vocabulary and adherence to it is an additional bonus of UML in building the Babylon towers of software architectures.

Figure 7.9 describes views of the models or diagrams that are defined in [OMG UML 2001].

These eight types of diagrams with different levels of detail and abstraction will capture your Software Architecture design in the UML model. Due to specifics of application domain and target software architecture, and often at your discretion, not all diagrams and not every fidelity level will be required in every project.

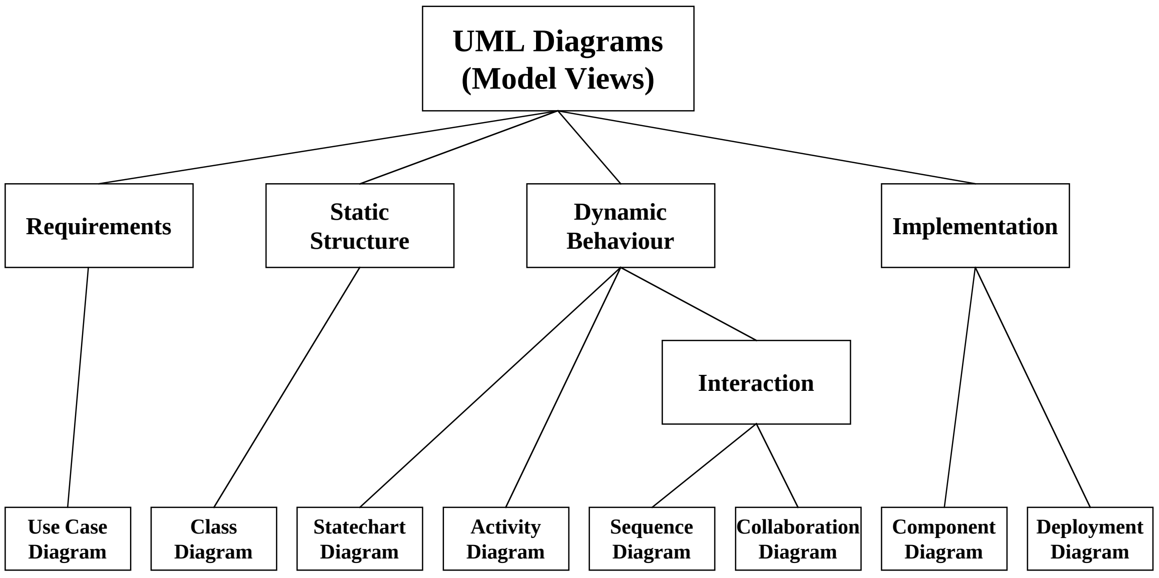


Figure 7.9. UML Diagrams, Canonical Taxonomy

We provide examples of some more common UML diagrams below.

Use Case diagram helps document functional requirements in the form of the real life scenario. *Use Case* defines some business process and identifies *Actors* involved. Each high-level *Use Case* may be drilled down on separate diagrams if required.

Figure 7.10 depicts *Use Cases* and *Actors* in the fictitious Soccer Team system. We introduced three *Actors* – *Manager*, *Coach* and *Player*. Obviously, our system is not modelled on the English Premier League – our *Manager* is a pretty hands-off type of guy and delegates all responsibilities of running the team to the *Coach*.

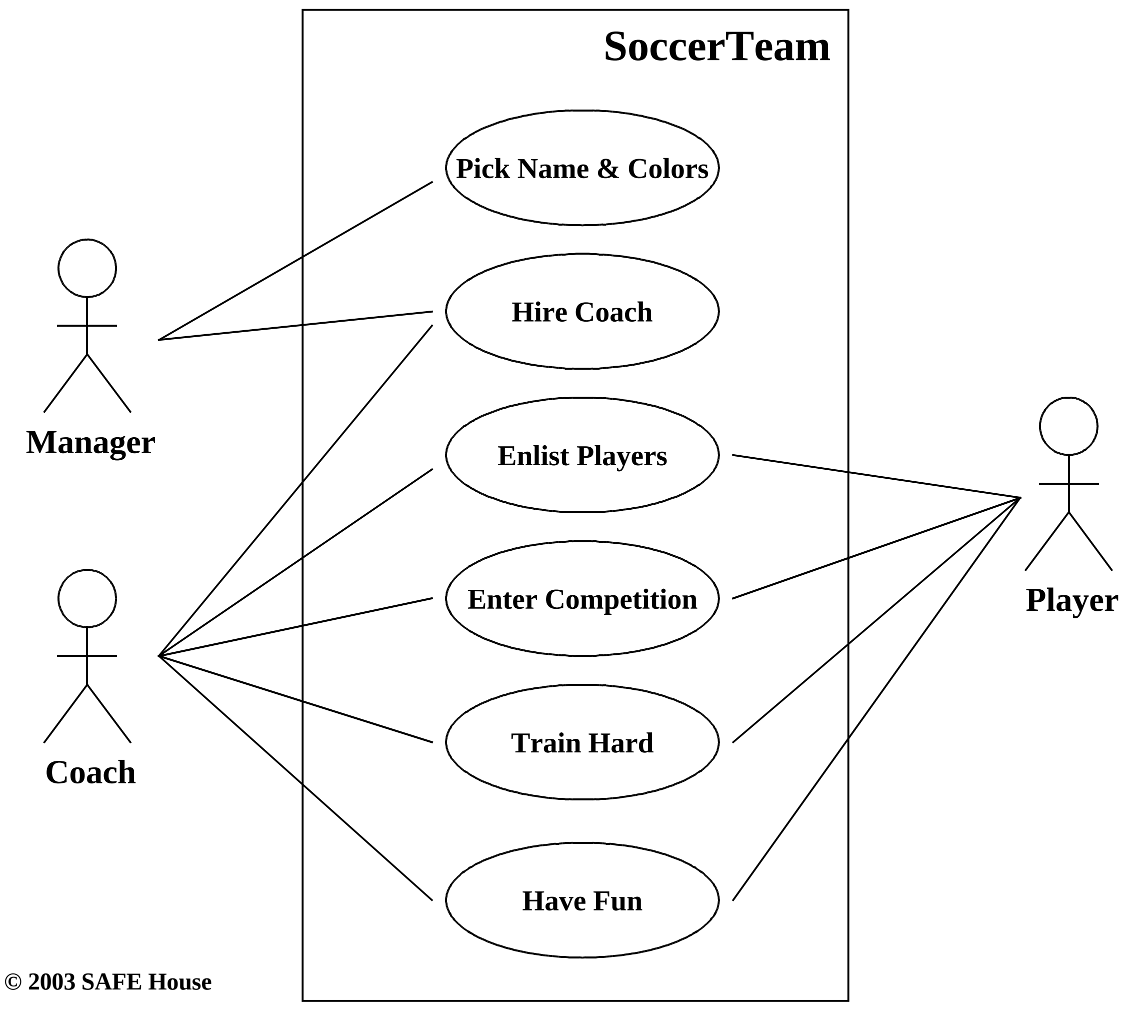


Figure 7.10. UML Use Case Diagram

Figure 7.11 shows a fragment of the *Class Diagram*. High-level *Class Diagram* may depict *Classes* as simple rectangle, without compartments for *attributes*, *operations*, *exceptions* etc. (as shown on the blow-out for the *Person* class).

SoccerTeam class is a 'kind of' *SportTeam* class. Hence, *SoccerTeam* inherits from (or sub-type of) the *SportTeam* class.

Instances of *Player* class and *Coach* class are 'part of' the *SoccerTeam* class. This relationship or association is depicted as *Aggregation*. Every instance of the *Player* class has a number in the *SoccerTeam* – diagram captures this fact in the adornment *{ordered}* for the *Player* in the association with the *SoccerTeam*.

Player and *Coach* are species of (or 'kind of') the type *Person*. However, we've chosen not to enforce inheritance here. One explanation could be that our system does not leverage behaviours of players and coaches as persons sufficiently enough to warrant the use of inheritance, but rather relies on some attributes of players as persons. Another explanation – we just wanted to show you more types of associations in action, and kind of decisions that object-oriented designer can make and justify.
We say, *Person* and *Coach* use or depend on the class *Person*.
Detailed Class Diagram for the *Player* class shows attribute *persDetail* of type *Person*. This way we avoided inheritance by including the characteristics of the *Player* as a *Person* into the *Player* class itself.
<<< Class Diagram for the *Player* class is not shown in this version due to conversion hickups >>>

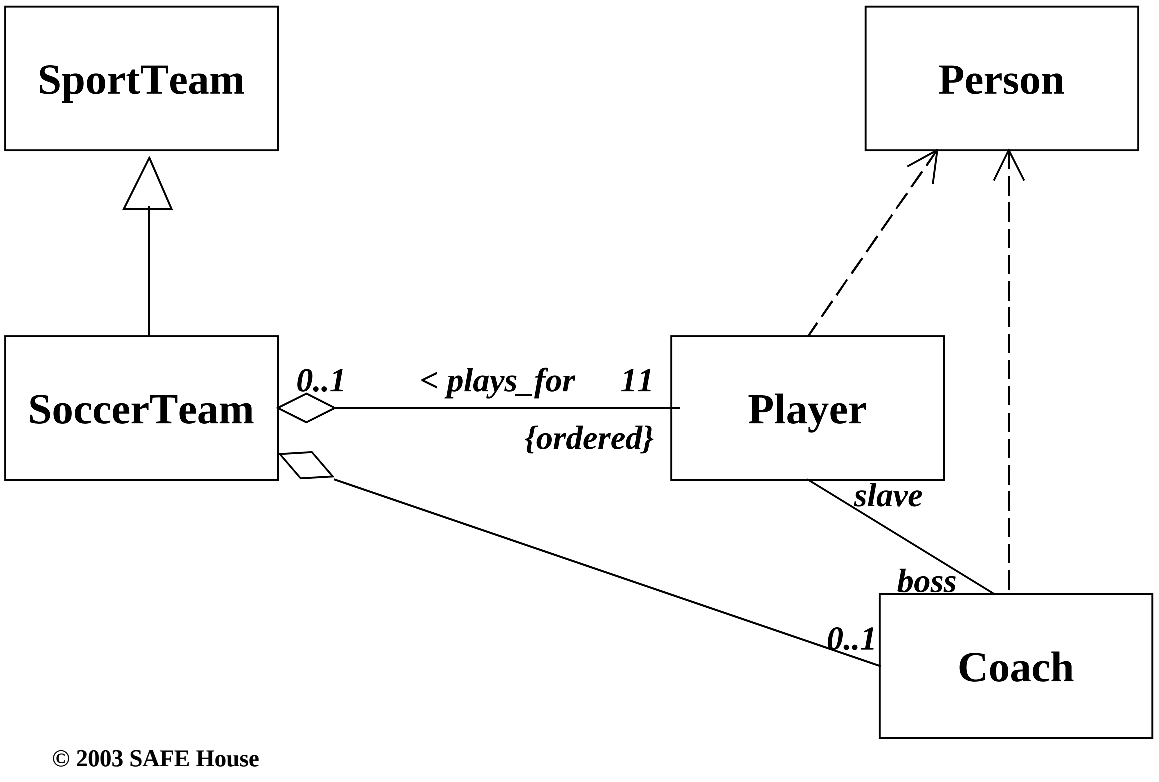


Figure 7.11. UML Class Diagram

Figure 7.12 provides *Sequence Diagram* for the *Use Case* “Create Season Fixture”.
Use Case diagram for the *SoccerTeam* application provided a higher-level view and did not show this use case. However, we assume that detailed *Use Case* design identified *Use Case* “Create Season Fixture” (among few others) when we drilled down *Use Case* “Enter Competition”.
‘Swimming line’ defines the actor (or participating entity, or stakeholder, or object) in the *Use Case* process. Object instance initiates the next step in the sequence by sending the *message* to (by invoking the *operation* on) another object. These *operations* must be defined in the class. UML tool like Rational Rose or Visio will spot the inconsistency in your UML design if you’ve made a mistake. Timeline in the *Sequence Diagram* flows from the top down.

<<< Sequence Diagram for the *Create Season Fixture* use case is not shown in this version due to conversion hickups >>>

<<< Figure goes here… >>>

Figure 7.12. UML Sequence Diagram

Figure 7.13 gives an example of the *Deployment Diagram* with physical configuration details of your application. This view allows you to approach the network connectivity issues, partitioning your application on different server boxes, installation, capacity planning, licensing costs, business continuity and security.
Deployment Diagram shows a 3-tier architecture. Business logic layer is implemented on J2EE Application Server WebSphere. Two boxes have been deployed running three instances of WebSphere.

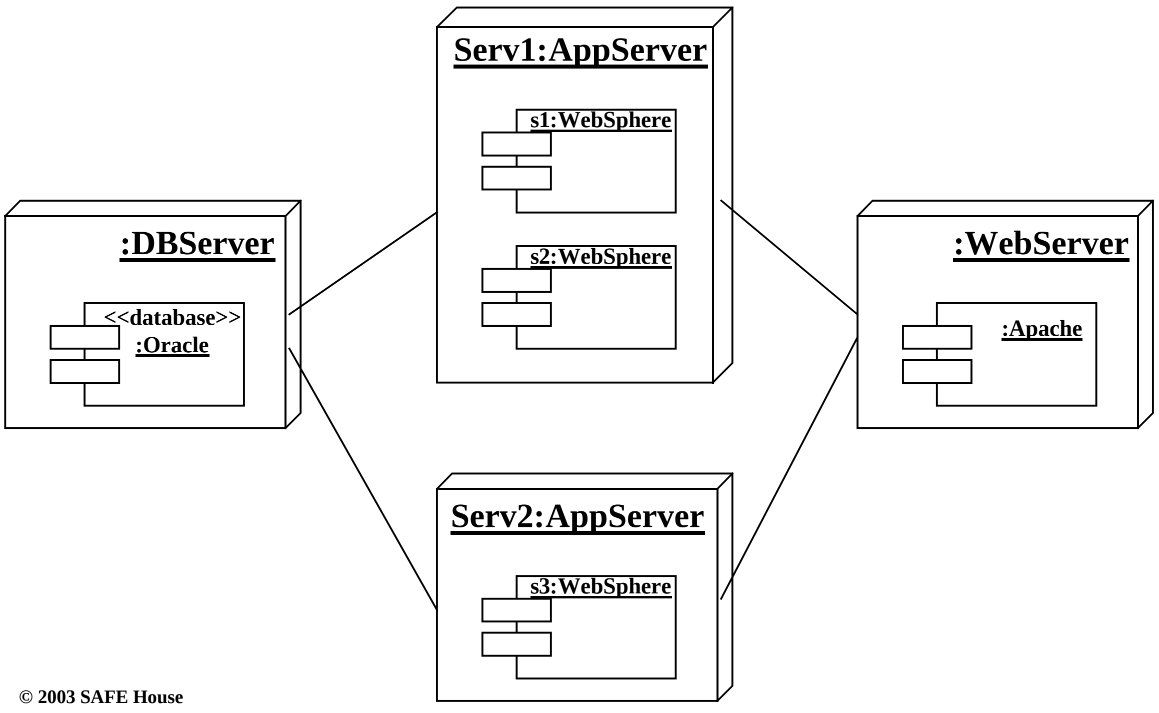


Figure 7.13. UML Deployment Diagram

Your project documentation will contain set of such diagrams covering all necessary entities, relationships and processes from different angles (views) to the required level of detail.

UML is a complex and evolving modelling language. Software Architects should see a lot of comfort and consolation in the fact that UML has reached a degree of maturity.

Complexity of the modelling language is well justified by our intent to capture and manage entities, relationships and processes in the infinitely complex real world.
However, 20% of UML features shall suffice in 80% of the real-life challenges.

No powerful modelling tool will ensure that you do not fall into trap of ‘analysis paralysis’. Your model should be as complex and detailed as required, but not more.

Rational Unified Process (RUP)

Rational Unified Process emphasises iterative and incremental nature of the object-oriented development.

Figure 7.14 presents a lifecycle of the Unified Process made up of four phases and nine process disciplines.
<<< Request permission >>>

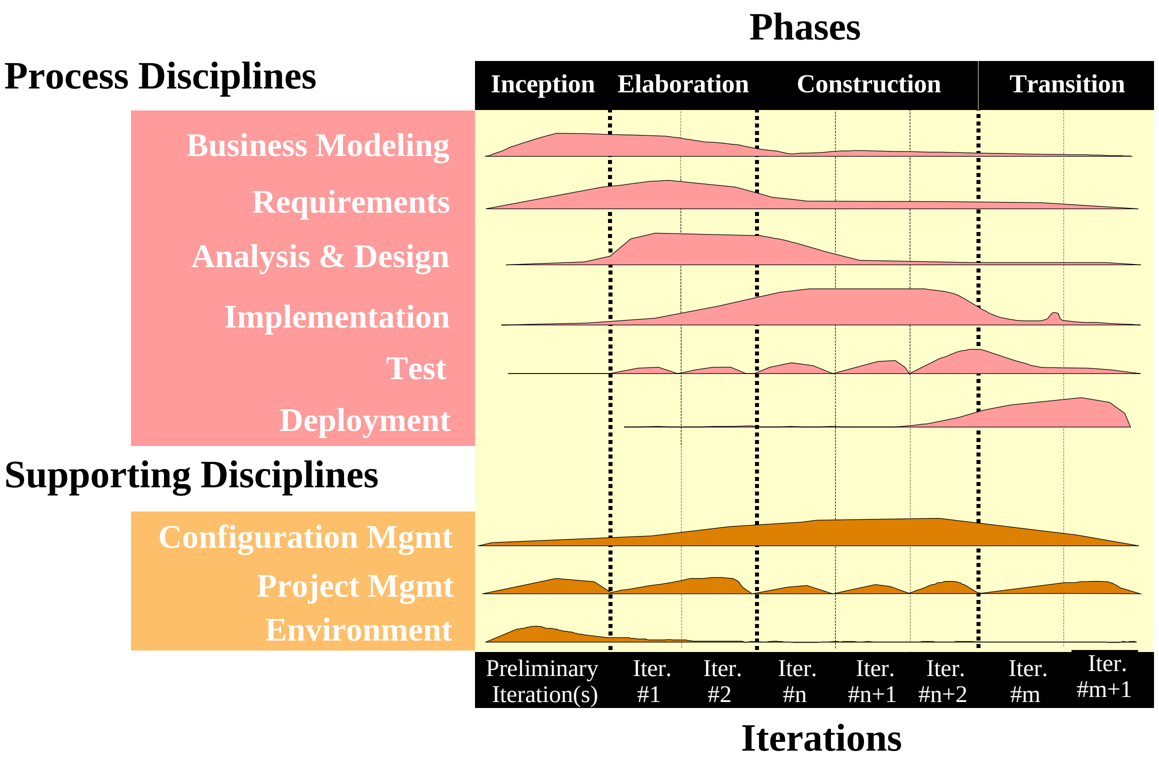


Figure 7.14. Rational Unified Process Lifecycle

Phases of the RUP are:

- ***Inception*** – definition of the project scope and the business case
- ***Elaboration*** – detailed analysis of the problem and architecture design
- ***Construction*** – detailed design and development
- ***Transition*** – implementation of the system

eXtreme Programming (XP)

Extreme Programming (XP) approach has been brought to life in direct response to inadequacies of ‘heavy’ or ‘monumental’ (Jim Highsmith) methodologies in the multitude of real-life projects.

Let’s hear from Kent Beck, the man himself: “XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements”.
And again, “XP is a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop software” [Beck 2000].

XP has its roots in Smalltalk development environment. XP was taking shape in the late 1980’s through refinement on numerous projects. One project especially provided a turning point in XP entering the mainstream – C3 payroll project at Chrysler.
XP was conceived and successfully promoted to the wider community by Kent Beck and Ward Cunningham. In essence, XP software development approach is *adaptive* (as opposed to *predictive* in heavy methodologies) and people-oriented.

Four pillars of XP methodology: Communication, Feedback, Simplicity and Courage. From this foundation, XP promotes several practices that XP projects should follow. Surprisingly, most of these practices are old as a world, yet often forgotten. Some practices are not so conventional and challenge many stale pre-conceptions of the software development.

XP emphasises incremental ubiquitous testing that is an intrinsic part of the coding process. Code becomes a primary source for documentation, and XP relaxed demands on documentation saved a lot of trees but also raised a lot of eyebrows. Cutting code is not a solitary process either – program

developers are working in pairs at the same terminal. Team constantly talks to the business owners who became a part of the development team. Incremental deliverables in a small manageable iteration do not provide any surprises. As a result, problems surface at the earliest possible moment, when fixing them does not cost an arm and a leg yet, and there is no temptation to cover them up.
Changes are most welcome as they ensure the deliverable to be as close to the true business requirements as possible.

XP methodology goes a long way toward identifying and meeting the true business requirements. This approach all but eliminates the possibility for the very familiar to IT practitioner situation when business requirements are not properly captured (provided we, together with business, are able to articulate them with sufficient rigour in the first place) and not satisfied. Requirements themselves will likely present a moving target in a very dynamic business environment.
In other words, XP approach to software development helps us to avoid the dreaded situation on the late stage of the project that was observed by the inspiration of ours, the great Dilbert: “It’s just what we asked for! But it’s not what we want”.

Agile Modelling (AM)

Methodology embodies best practices and guidelines for the software development. Methodology is a ‘yellow brick road’ that supposed to bring us to the goals of our project.
How you reconcile these expectations for the methodology with the fact that every project is bound to be somewhat specific in a somewhat different context? How you adapt the heavyweight methodology to the fast-changing business environment so it responds quickly, does not live the life of its own, does not spin the wheels and does not create scary black hole for the resources and money, without the tangible feedback to the business?

IT practitioners themselves point to ‘the misguided efforts of theoreticians, the decades of cultural entropy within information technology departments, the marketing rhetoric of software development tools companies’ [Ambler 2001b].
Crisis in the software methodologies and their application have reached the point when there is need ‘to restore credibility to the world’.

[AgileAlliance] tells us the true story that happened with two of its founding members:
Ken Achwaber (a proponent of SCRUM) told of his days of selling tools to automate comprehensive, ‘heavy’ methodologies. Impressed by the responsiveness of Ken’s company, Jeff Sutherland (SCRUM) asked him which of these heavy methodologies he used internally for development. “I still remember the look on Jeff’s face,” Ken remarked, “when I told him, ‘None – if we used any of them, we’d be out of business!”

On February 2001, at the ski resort lodge in Wasatch Mountains of Utah, 17 people (all recognised methodology gurus) met to talk, relax and to find the common ground.
What emerged was the Agile Software Development Alliance [AgileAlliance] [WWW, AgileModelling].
This ‘gathering of organisational anarchists’ produced a *Manifesto for Agile Software Development* – signed by all participants.

By contrasting the new approach with ‘heavyweight’ methodologies, we could fall into trap of calling it a ‘lightweight’ – which implies cutting corners, creating somewhat abridged terse version of complete, rigorous and ‘correct’ methodology. These connotations of adjective ‘light’ would miss the point and would be completely misleading.
Participants coined term ‘agile’ – to highlight a no-nonsense nature of the new approach, its back-to-basics spirit, to put ability for the responsiveness to the business needs first, where it belongs.

Let’s quote from *The Agile Manifesto* [AgileAlliance]. <<<Request permission>>>

The Agile Manifesto Purpose:

“We are uncovering better ways of developing software by doing it and helping others do it. We value:
- *Individuals* and *interactions* over processes and tools

- *Working software* over comprehensive documentation
- *Customer collaboration* over contract negotiation
- *Responding to change* over following a plan

That is, while we value the items on the right, we value the items on the left more.”

The Agile Manifesto Principles:

“We follow the following principles:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
- Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to a shorter timescale
- Business people and developers work together daily throughout the project
- Build project around motivated individuals. Give them the environment and support they need, and trust them to get the job done
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
- Working software is primary measure of progress
- Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely
- Continuous attention to technical excellence and good design enhances agility
- Simplicity – the art of maximizing the amount of work not done – is essential
- The best architectures, requirements and designs emerge from self-organizing teams
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly“

Although not a revolution on its own (eg. XP is an agile process), Agile Modelling honestly deals with deficiencies of ‘heavyweight’ methodologies and does shatter some well-entrenched dogmas.

All too often, while building complex architectures in changing environment, we failed to see forest behind the trees.
Documentation does not automatically mean a good quality design; requirements refuse to get frozen at will; frozen design too may become irrelevant as our growing understanding gets captured in the software, but not in the design.

‘Heavy’ methodologies aimed to streamline software development process, to level it and make its quality more manageable and predictable, less dependable on whims of computer geeks of old – a very noble goal indeed.
Unfortunately, formal processes themselves gave ample opportunity for abuse and hiding the lack of professionalism in reams of project documentation – due to disconnect between real business needs and the system under construction, between rigid design and delivered software.
Agile Modelling does not seek simplicity at the expense of broad and deep knowledge of enterprise architecture and technologies. On the contrary, you can afford to omit some details only if you are fluent with the Enterprise Architect’s vocabulary, if you can see the big picture in terse blueprints, and team members can catch the ball all the time.

Complex enterprise architectures rarely give us a chance of working out all the details upfront in our designs, and then build the system to these perfect designs – recipe for the ‘”analysis paralysis” - the fear of moving forward until your models are perfect’ [Ambler 2001b].
Benefits of the iterative software development processes (as opposed to waterfall approach) have been recognised long before, and implemented in methodologies like RUP.
Agile Modelling takes iterative process of software construction to the next logical level – change becomes the norm. Every iteration, or rather increment, closes the loop with customer; customer is always right.

Although several well-known existing methodologies started calling themselves ‘agile’, Agile Modelling may create a culture shock in many IT departments. In order to address existing pains

without introducing the new headaches, Agile Modelling proponents should pay special attention to the robust Change Management processes in the organization.
After all, if you are about to raise the wave of iterations and changes, and to make changes a norm, you better make sure you can manage changes.

IT practitioners will do themselves a favour by watching this space.

When you should consider going ‘light’ (or agile, or adaptive, as opposed to ‘heavy’, or predictive, or ‘monumental’, or planned)?
Martin Fowler [WWW, Fowler, ‘The New Methodology’ web paper] advises that “adaptive approaches are good when your requirements are uncertain or volatile”, and when team is not too big.
“What if you have both a large project and changing requirements? I don’t have a good answer to this… I can tell you that things will be very difficult, but I suspect you already know that.”
Hardly a disappointing surprise – there is no silver bullet.

You take the adaptive route if you can rely on and trust to your developers, if they are skilled and motivated.

<<< Request permission >>>
“So, to summarise. The following factors suggest an adaptive process:

- Uncertain or volatile requirements
- Responsible and motivated developers
- Customer who understands and will get involved.

These factors suggest a predictive process:

- A team of over fifty
- Fixed price, or more correctly a fixed scope, contract.”

Capability Maturity Model Integration (CMMI)

Software Engineering Institute (SEI) at Carnegie Mellon University proposed Capability Maturity Model (CMM) as a framework from which a process for large complex software efforts can be defined.

First, a little bit of history and context that will help you grasp better the core concepts, goals, applicability and the near future of this evolving methodology.

Since 1950s, when Information Technology started to penetrate the very fabric of human society (including business, defence, communication, utilities, health, academic research, infotainment etc) software industry and its beneficiaries became increasingly dissatisfied with unreliable and unpredictable nature of software.
In the initial excitement of seeing computers somehow doing some work in the first place, users and software providers alike needed to rely on software in conducting their mission-critical core business activities, and became less and less forgiving for the lack of quality and functionality, for the overrun of budgeted costs and time schedules.
Quality measurement and quality management of the software products and processes have steadily risen in importance and have become a high priority in IT in 1980s.

In 1986, the Software Engineering Institute (SEI), with assistance from the Mitre Corporation, began developing a software process maturity framework. This project was largely driven by the needs of software development and acquisition in defence and large corporations, but with rippling effect impacting the whole of IT industry.
SEI evolved the maturity framework into Capability Maturity Model for Software (CMM). CMM v0.6 was released in 1990, and refined into CMM v1.0 in 1991.
Review and feedback from the experiences in the software community culminated in CMM v1.1 in 1993.

CMM continued to progress towards CMM v2 in 1997. However, in October 1997, Department of Defence (SEI’s sponsor) recognised new software industry imperatives for reconciling the multiple CMMs, and directed that the Software CMM v2 release be halted in favour of work on CMM Integration (CMMI).

CMMI is to become a successor of CMM.

CMM and CMMI are very similar in high-level goals and concepts, but differences in their terminology and the model structure may cause some confusion. CMMI specifications address terminology and model structure evolution in detail.
Main thrust in both CMM and CMMI remains the same – measurable and manageable quality of the software process, and defining the path for the evolutionary software process improvement through well-defined maturity levels.

CMM structure was defined in terms of five Maturity Levels that contain pre-defined Key Process Areas (KPA).
KPA identifies activities aiming to achieve a set of goals considered important for the process capability of this Maturity Level. KPA further organized by Common Features and Key Practices.

CMMI starts off with introduction of distinction (and, accordingly, the need for making first choices) between *continuous* and *staged* representation of the model, and between the disciplines of *Systems Engineering* and *Software Engineering*. This distinction in CMMI models brings about rather unfortunate differences in the CMMI models structure. We will ignore these differences in CMMI models for the purpose of our discussion as much as possible (this may not be easy as you delve into details – eg, *continuous* representation of CMMI model actually has six Maturity Levels, and they named slightly different to the ones in the *staged* representation – life was not meant to be easy).
Like CMM, CMMI structure is based on Maturity Levels. Unlike CMM, CMMI Maturity Levels contain Process Areas (PA), not KPAs. CMMI Process areas aim to achieve *Specific Goals* or *Generic Goals* by implementing (you might have guessed) the *Specific Practices* or *Generic Practices*.
In case of confusion in dissecting the CMMI model structure rigorously, we use staged representation as default.

CMMI Generic Practices are grouped into following Common Features:

- Commitment to Perform
- Ability to Perform
- Directing Implementation
- Verifying Implementation

Both in CMM and CMMI, Maturity Level is a well-defined evolutionary plateau towards achieving a mature software process by lifting the process capability from one level to the next.

	CMM Maturity Level	CMMI Maturity Level	Characteristics
1	Initial	Initial	The software process is *ad hoc*, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics. Success cannot be repeated unless the same experienced individuals are assigned to the next project. Organization can produce products that work. However, they often greatly exceed the budget and schedule of the project
2	Repeatable	Managed	Basic project management processes are established to track cost, schedule and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications
3	Defined	Defined	The software process for management and development activities is documented, standardized and integrated into a standard software process for your organization. All projects use an approved, tailored version of the process. Standard processes are consistently applied across the whole enterprise
4	Managed	Quantitatively Managed	Detailed measures, called metrics, of the software process and product quality are collected. Both the software process and products are quantatively understood and controlled. Processes exhibit predictable performance
5	Optimising	Optimising	Continuous process improvement is enabled by quantative feedback from the software processes and from piloting innovative ideas and technologies. Processes adapt to guarantee achievement of quantitative performance improvement objectives

Rigorous software processes (as opposed to sloppy processes or a free-fall scenario) significantly improve chances of completing projects on time, on budget and to the customer satisfaction.
Software Services vendors who are certified to CMM Level 4 or 5, are able to achieve this dream in 85% of projects (and be not far away on the rest of the projects).
'Normal' failure rates for the most of other mere mortal integrators are estimated at around 30%. Other unflattering estimations point out that we belong to the 'industry with an 85% failure rate' [Ambler 2001a, p.7].

Author had enlightening experiences of dealing with CMM Level 5 certified Software Services Vendor, and can attest that rigours in the enterprise software processes do make a difference.

Patterns and Frameworks

Pattern movement in software construction has been inspired by the work of Christopher Alexander – urban designer and ('real') building architect.
Kent Beck and Ward Cunningham first observed the similarity between software patterns and building architecture patterns [Cope 1996].

'Each pattern is a three-part rule, which expresses a relation between a certain *context*, a *problem*, and a *solution*' [Alexander 1979].

Following its architectural heritage, software patters use the notion of *force*. However, term *force* used in software patterns figuratively, as there is no physical force to deal with directly (like forces of gravity or forces from adjoining structures).
Patterns aim to identify *forces* that determine the *problem* in the *context*. *Solution* is to be found by balancing *forces*.

Good pattern must provide a mature, proven solution. Patterns imply repetition, re-use – we should be able to apply them over and over again (three times at least – Rule of 3) to find a successful solution to the problem in different contexts.

In one sense, pattern is just documentation, a literary form.
If a pattern is literature, it is like a play in that the Solution should provide catharsis. The Context introduces the "characters" and the "setting"; the Forces provide a plot, and the Solution provides a resolution of the tension built up in the conflict of the Forces [Cope 1996].

Again, if a pattern is literature, and pattern designer is writer – there are seven habits that effective pattern writer should adopt [Vlissides 1996]:

- Habit 1: Taking Time to Reflect
- Habit 2: Adhering to a Structure
- Habit 3: Being Concrete Early
- Habit 4: Keeping Patterns Distinct and Complementary
- Habit 5: Presenting Effectively
- Habit 6: Iterating Tirelessly
- Habit 7: Collecting and Incorporating Feedback

Another notion borrowed into software patterns from the building architecture is the *Pattern Language*.
Pattern Language – collection of patterns that build on each other to generate a system [Cope 1996].

Patterns often (quite arbitrarily) categorised or layered in three levels of abstraction – *idioms*, *design patterns* and *frameworks* [Cope 1996]:

- *Idioms* – low-level patterns that depend on a specific implementation technology
- *Design Patterns* – made their debut in the landmark book [GoF 1995]. Design Patterns capture the good practices of object-oriented design independent of a particular programming language. They are micro-architectures – structures larger than objects but not large enough to be system-level organizing principles
- *Framework Patterns* – a *framework* is a partially completed body of code designed to be extended for a particular application

Framework provides an overall infrastructure that gives an architect ability to plug-n-play with components.

Aesthetics and the professional ethics are becoming important tangible factors in designing and applying software patterns.
Potentially, due to subjective nature of software quality and to the often-unclear limits of software intellectual property, patterns are open to abuses of unproven or unscrupulous techniques.

Patterns and Frameworks is a playing field for mature IT practitioners who are able to find a right balance between novelty and conformism, in ethical and creative environment. Cultural norms of some innovative R&D organizations may be tested by the aggressive “disregard for originality” (Brian Foote) of the pattern community.
However, do not get fooled by these self-deprecating witty statements of the software patterns pioneers. Pattern movement brings the nuggets of highest professionalism, deep understanding of software architectures and empirical wisdom to the wider IT community.
Introduction of the higher discipline and re-use of the best practices in the software design is an innovation and a huge challenge of its own.

<<< … >>>