CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Lecture 5: Implementing Variability

Gregory Gay
TDA 594/DIT 593 - November 16, 2021

Variability

- **The ability to derive different products from a common set of assets.**
- Implementation: *How* do we build a custom product from a feature selection?
  - Binding Time
  - Technology (Language vs Tool-Based Implementation)
  - Representation (Annotation vs Composition)

Today's Goals

- Basic implementation concepts
- Tool-based Implementation
  - Preprocessors, Build Systems, Version Control
- Introduce language-based implementation
  - Parameters

Binding Time

- Compile-time Binding
  - Decisions made during compile.
  - #IFDEF preprocessor in C/C++.
- Load-time Binding
  - Decisions made when program starts.
  - Configuration file or command-line flags.
- Run-time Binding
  - Decisions made while program runs.
  - Method or API call.

```
class Node {
    int id = 0;

    //#ifdef NAME
    private String name;
    String getName() { return name; }
    //#endif
    //#ifdef NONAME
    String getName() { return String.valueOf(id); }
    //#endif

    //#ifdef COLOR
    Color color = new Color();
    //#endif

    void print() {
        //#if defined(COLOR) && defined(NAME)
        Color.setDisplayColor(color);
        //#endif
        System.out.print(getName());
    }
}
//#ifdef COLOR
class Color {
    static void setDisplayColor(Color c){/*...*/}
}
//#endif
```

```
C19ZRMR:Downloads ggay$ cat review.txt | cut -d" " -f 1 | head -1
View
C19ZRMR:Downloads ggay$ cat review.txt | cut -d" " -f 1-5 | head -1
View Reviews
```

```
if (type.equals("cheese")){
    pizza = new CheesePizza();
else if(type.equals("pepperoni")){
    pizza = new PepperoniPizza();
}
```

Binding Time

- Compile-time binding improves performance.
  - … but executable cannot be reconfigured.
- Load-time binding configured at execution.
- Run-time binding can be configured any time.
  - … but reduced performance/security, increased complexity.

Technology

- Language-based Implementation
  - Use programming language mechanisms to implement features and derive product.
  - Pass parameters at run-time.
- Tool-based Implementation
  - Use external tools to derive a product.
  - Use preprocessor to compile only the requested features.

Technology

- Language-Based Implementation
  - Feature implementation **and** management in code.
  - Easy to understand.
  - Feature management/boundaries easily vanishes.
- Tool-Based Implementation
  - Separate implementation and management.
  - Simplifies code.
  - Must reason about multiple artifacts.

Annotation-Based Representation

- Code in common code base.
- Code related to a feature is marked.
  - Preprocessor annotations, if-statements.
- Code belonging to deselected features:
  - ignored (load-time, run-time)
  - removed (compile-time).
- Adds complexity, reduces modularity/readability.

Composition-based Representation

- Feature code in dedicated location.
  - Class, file, package, service
- Selected units combined to form product.
- Requires clear mapping between features and units
- Can combine annotation and composition.
  - Annotation-based approaches remove code.
  - Composition-based approaches add code.

Some Examples

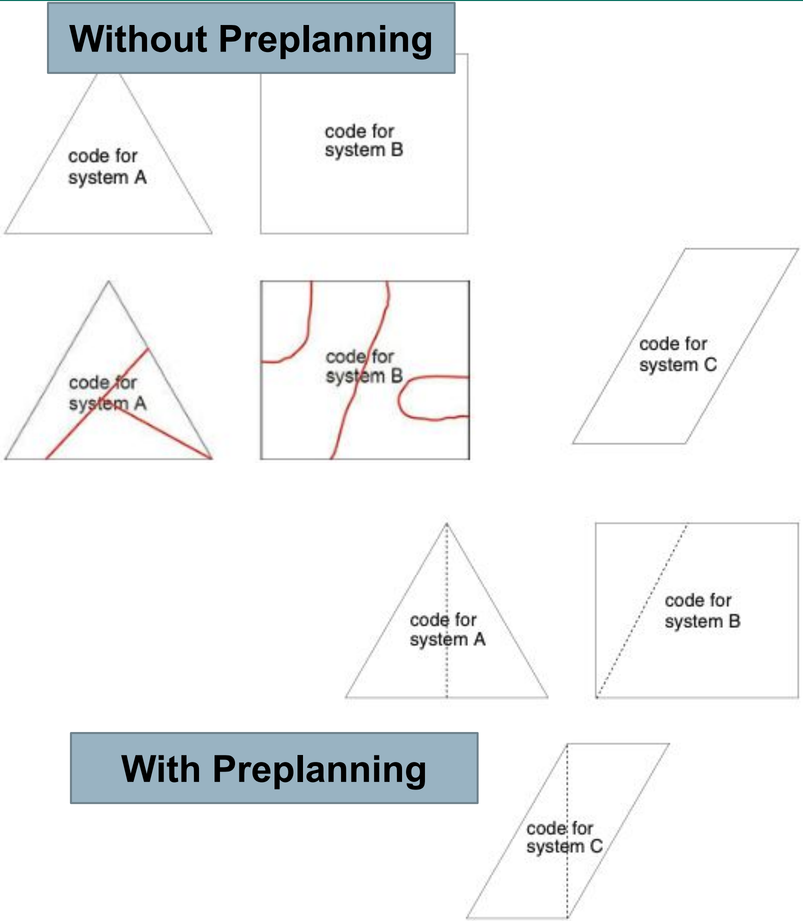
Preprocessors	Compile-Time	Tool-Based	Annotation-Based
Build Systems	Compile-Time	Tool-Based	Composition-Based
Parameters	Load or Run-Time	Language-Based	Annotation-Based
Design Patterns	Load or Run-Time	Language-Based	Composition-Based
Frameworks	Load or Run-Time	Language-Based	Composition-Based
Components	Any	Any	Composition-Based

Quality Criteria

- We want a SPL to have:
  - Low preplanning effort
  - Feature traceability
  - Separation of concerns
  - Information hiding
  - Granularity
  - Uniformity
- These often conflict!

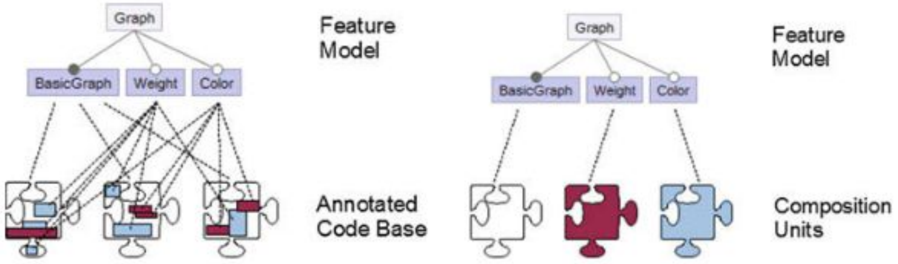
Preplanning Effort

- Preplanning is required to enable code reuse.
- Implementation techniques
  - Can minimize the need for extensive preplanning.
  - Can support change and addition of features.



Feature Traceability

- Ability to link a feature from the problem space (theoretical model) to the solution (code).



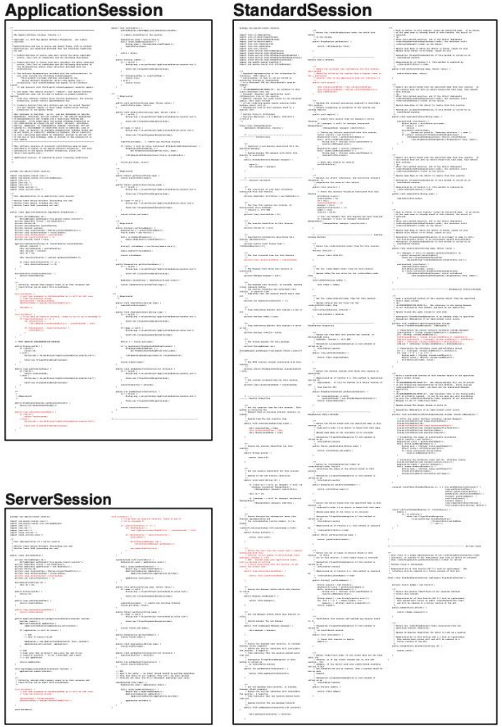
  - Very important to ensuring correct implementation.
  - Preprocessor directives are easier to detect than run-time parameters (if-statements).
  - Easiest to trace if feature code is contained to a single unit, harder if code is spread across units.

Separation of Concerns

- Development should be structured into concerns (focuses) that are implemented separately.
  - Ignoring irrelevant details.
  - In a SPL, features are the concerns.
- Features separated into distinct artifacts are easier to debug and maintain.
  - Code structures with *high cohesion* only contain highly related code.

Cross-Cutting Concerns

- May be difficult to separate features.
  - Cross-cutting concerns are features that span multiple units (classes).
  - **Code Scattering** - feature code appears across multiple other concerns.
  - **Code Tangling** - code of two features directly mixed.



Cross-Cutting Concerns

```
class Graph {
  Vector nv = new Vector(); Vector ev = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nv.add(n); nv.add(m); ev.add(e);
    if (Conf.WEIGHTED) e.weight = new Weight();
    return e;
  }
```

Code Replication

```
    e.weight = w; return e;
  }
  void print() {
    for(int i = 0; i < ev.size(); i++)
      { ((Edge)ev.get(i)).print();
    }
  }
}
```

```
class Color {
  static void setDisplayColor(Color c) { ... }
}
```

```
class Node {
  int id = 0;
  Color color = new Color();
  void print() {
   if (Conf.COLORED) Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
class Edge {
  Node a, b;
  Color color = new Color();
  Weight weight;
  Edge(Node _a, Node _b) { a = _a; b = _b; }
  void print() {
    if (Conf. COLORED) Color.setDisplayColor(color);
    a.print(); b.print();
    if (!Conf.WEIGHTED) weight.print();
  }
}
```

```
class Weight { void print() { ... } }
```

Cross-Cutting Concerns

- Scattering leads to hidden concerns.
  - Hard to find all feature code.
  - Hard to coordinate developers.
  - Hard to evolve code.
- Some cross-cutting concerns are required.
  - Important to minimize number, track ones that exist.

Information Hiding

- Divide each module into internal and external parts:
  - Internal (Secret): Bulk of code
  - External: Interface that surfaces accessible functions
- A module can be understood by examining its contents and only the interfaces of other modules.
  - Simplifies and un-biases development.
  - Allows independent teams to develop features.

Information Hiding

- Key challenge is to design small, clear interfaces.
  - Makes communication explicit.
  - Enables more hiding of information.
- Enables separation of concerns.
  - Good separation of concerns enables information hiding.
  - Requires both… which requires pre-planning.

Granularity

- Implementing a feature may require code changes
  - Coarse-grained: A new Java class
  - Fine-grained: Adding statements to an existing function.
- Implementation mechanisms define how code can be easily changed.
  - Composition-based: Supports coarse-grained changes.
  - Annotation-based: Supports fine-grained changes.

Uniformity

- Features can be implemented in different languages or formats.
- Product line implementation techniques should encode and process artifacts in a ***uniform*** manner.
  - It should not matter if code was written in C++ or Java, we should be able to work with it in the same way.

Tool-Based Implementation

Preprocessors

- Optimize code before compilation.
  - Often used by compilers to produce faster executable.
  - Can selectively include or exclude code.
- Most famous - cpp
  - “The C Preprocessor” (C, C++)
- Exist for many languages.

```
class Node {
    int id = 0;

    //#ifdef NAME
    private String name;
    String getName() { return name; }
    //#endif
    //#ifdef NONAME
    String getName() { return String.valueOf(id); }
    //#endif

    //#ifdef COLOR
    Color color = new Color();
    //#endif

    void print() {
        //#if defined(COLOR) && defined(NAME)
        Color.setDisplayColor(color);
        //#endif
        System.out.print(getName());
    }
}
//#ifdef COLOR
class Color {
    static void setDisplayColor(Color c){/*...*/}
}
//#endif
```

Implementation with cpp

- `#include` enables import from another file.
  - `#include <string.h>`
- `#define` used to substitute value for reference.
  - Reserve one per feature.
  - `#define FEATURE_NAME TRUE`
    - (if the feature is selected, don’t use `#define` if not selected)
- `#ifdef/#endif` used to conditionally include code.
  - `#ifdef FEATURE_NAME`

Implementation with cpp

```
class Node {
    int id = 0;

    //#ifdef NAME
    private String name;
    String getName() { return name; }
    //#endif
    //#ifdef NONAME
    String getName() { return String.valueOf(id); }
    //#endif

    //#ifdef COLOR
    Color color = new Color();
    //#endif

    void print() {
        //#if defined(COLOR) && defined(NAME)
        Color.setDisplayColor(color);
        //#endif
        System.out.print(getName());
    }
}
//#ifdef COLOR
class Color {
    static void setDisplayColor(Color c){/*...*/}
}
//#endif
```

- `#ifdef`
- `#if defined(MACRO)`
  - Check if a macro is defined. If true, code is included.
  - Define macro for included features.
- `#if (...)` can check a user-defined condition.

Implementation with cpp

```
1  static int __rep_queue_filedone(dbenv, rep, rfp)
2    DB_ENV *dbenv;
3    REP *rep;
4    __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6    COMPQUIET(rep, NULL);
7    COMPQUIET(rfp, NULL);
8    return __db_no_queue_am(dbenv);
9  #else
10   db_pgno_t first, last;
11   u_int32_t flags;
12   int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14   DB_MSGBUF mb;
15 #endif
16   // over 100 lines of additional code
17 #endif
18 }
```

- `#ifndef`
  - “if not defined”
- `#else`
- Note nesting of directives.
  - Line 17 ends line 5 directive.

Implementation with Antenna (Java)

- Similar to cpp
  - Annotations written as comments.
  - Comments out code that is not selected and uncomments code that is selected.
- Available from http://antenna.sourceforge.net/
  - Part of FeatureIDE or can used from command line.

Implementation with Antenna (Java)

- Annotate code using comments:
  - `//#if FEATURE_NAME`
    - If FEATURE_NAME is chosen, include this code.
  - `//#elif OTHER_FEATURE`
    - else if OTHER_FEATURE chosen, include this code.
  - `//#else`
  - `//#endif`
- Instead of removing lines, Antenna comments out lines, inserting `//@`

Examples

(Hello, Beautiful, World)

```java
public class Main {
  public static void main(String[]
        args) {
    //#if Hello
    System.out.print("Hello");
    //#endif
    //#if Beautiful
    System.out.print(" beautiful");
    //#endif
    //#if Wonderful
//@ System.out.print(" wonderful");
    //#endif
    //#if World
    System.out.print(" world!");
    //#endif
  }
}
```

(Hello, Wonderful, World)

```java
public class Main {
  public static void main(String[]
        args) {
    //#if Hello
    System.out.print("Hello");
    //#endif
    //#if Beautiful
//@ System.out.print(" beautiful");
    //#endif
    //#if Wonderful
    System.out.print(" wonderful");
    //#endif
    //#if World
    System.out.print(" world!");
    //#endif
  }
}
```

Let’s take a break!

Proper Use of Preprocessors

- Should wrap around an entire function, declaration, or expression.

```c
#if defined(__MORPHOS__) &&
      \defined(__libnix__)
extern unsigned long *__stdfiledes;

static unsigned long
    fdtofh(int filedescriptor) {
  return __stdfiledes[filedescriptor];
}
#endif
```

```c
void tcl_end() {
#ifdef DYNAMIC_TCL
  if (hTclLib) {
    FreeLibrary(hTclLib);
    hTclLib = NULL;
  }
#endif
}
```

- Bad annotations wrap partial expressions.

```c
  int n = NUM2INT(num);
#ifndef FEAT_WINDOWS
  w = curwin;
#else
  for (w = firstwin; w != NULL;
      w = w->w_next, --n)
#endif
    if (n == 0)
      return window_new(w);
```

```c
  if (!ruby_initialized) {
#ifdef DYNAMIC_RUBY
    if (ruby_enabled(TRUE))
#endif
      ruby_init();
```

```c
int put_eol(fd)
    FILE *fd;
{
  if (
#ifdef USE_CRNL
      (
#ifdef MKSESSION_NL
        !mksession_nl &&
#endif
        (putc('\r', fc) < 0)) ||
#endif
        (putc('\n', fd) < 0))
    return FAIL;
  return OK;
}
```

Benefits of Preprocessors

- Easy to learn (annotate and remove code).
- Can be applied to code and other artifacts.
- Allow changes at any level of granularity.
- Easy to map features and code.
- Can be added to a non-product line to transform it into one over time.

Drawbacks of Preprocessors

- Feature code scattered across codebase and mixed with other features.
- Encourage developers to patch and add to code instead of refactoring.
- Can make it hard to understand control flow in code
- Can introduce errors, especially when used on partial statements.

Build Systems

- Schedules and executes build-related tasks.
  - Compilation, testing, packaging, etc.
  - Ex: Make, Maven, Gradle
- Can be used to manage compile-time variability.

```
<?xml version = "1.0"?>
<project name = "Hello World
Project" default = "info">
   <target name = "info">
      <echo>Hello World - Welcome
       to Apache Ant!</echo>
   </target>
</project>
```

Variability in Build Scripts

- Compiles code conditionally depending on features selected.
  - Feature selection read from file or inferred from environment (language, location, software).
  - Features can control how files compiled.

```
#!/bin/bash -e

rm *.class
javac Graph.java Edge.java Node.java \
      Color.java
jar cvf graph.jar *.class
```

```
#!/bin/bash -e

if test "$1" = "--withColor"; then
  cp Edge_withColor.java Edge.java
  cp Node_withColor.java Node.java
else
  cp Edge_withoutColor.java Edge.java
  cp Node_withoutColor.java Node.java
fi

rm *.class
javac Graph.java Edge.java Node.java
if test "$1" = "--withColor"; then
  javac Color.java
fi

jar cvf graph.jar *.class
```

Example - Linux Kernal

- Kbuild decides which files to compile based on feature selections.
  - `obj-y += foo.o`
    - Compile and link `foo.c`.
  - `obj-m += foo.o`
    - Build `foo.c` as loadable module.
  - `lib-y += foo.o`
    - Include `foo.c` as a library.
  - `obj-(COFIG_FOO) += foo.o`
    - (CONFIG_FOO) is a feature. Set to (y, m, n) for compile, module, skip.

```
#
# Makefile for the video capture/playback device drivers.
#

tuner-objs      :=      tuner-core.o

videodev-objs   :=      v4l2-dev.o v4l2-ioctl.o v4l2-device.o

obj-$(CONFIG_VIDEO_DEV) += videodev.o v4l2-int-device.o
ifeq ($(CONFIG_COMPAT),y)
  obj-$(CONFIG_VIDEO_DEV) += v4l2-compat-ioctl32.o
endif

obj-$(CONFIG_VIDEO_V4L2_COMMON) += v4l2-common.o

ifeq ($(CONFIG_VIDEO_V4L1_COMPAT),y)
  obj-$(CONFIG_VIDEO_DEV) += v4l1-compat.o
endif

obj-$(CONFIG_VIDEO_TUNER) += tuner.o
obj-$(CONFIG_VIDEO_TVAUDIO) += tvaudio.o
obj-$(CONFIG_VIDEO_TDA7432) += tda7432.o
obj-$(CONFIG_VIDEO_TDA9875) += tda9875.o

...

EXTRA_CFLAGS += -Idrivers/media/common/tuners
```

Discussion

- Build systems are language agnostic (uniform).
- Does not require extensive preplanning.
  - But no notion of consistency or modularity.
- Good if features can be mapped to files.
  - Must replace entire file, so best if feature code mapped to single class placed in its own file.
- Executes other variability mechanisms
  - Run pre-processors, select branch from version control, create configuration file.

Parameter-Based Implementation

Language-Based Variability

- Programming languages offer means to implement variability in different ways.
  - if-statement offers a choice between two options.
- Common approaches:
  - Parameters
  - Design Patterns
  - Frameworks and Libraries
  - Components and Services

Parameter-based Implementation

- Use conditional statements to alter control flow based on features selected.
- Boolean variable based on feature, set globally or passed directly to methods:
  - From command line or config file (load-time binding)
  - From GUI or API (run-time binding)
  - Hard-coded in program (compile-time binding)

```java
class Conf {
  public static boolean COLORED = true;
  public static boolean WEIGHTED = false;
}


class Graph {
  Vector nodes = new Vector();
  Vector edges = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n,m);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    if (Conf.WEIGHTED)
      e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    if (!Conf.WEIGHTED)
      throw new RuntimeException();
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
  void print() {
    for(int i=0; i<edges.size(); i++){
      ((Edge) edges.get(i)).print();
      if(i < edges.size() - 1)
        System.out.print(" , ");
    }
  }
}
```

```java
class Node {
  int id = 0;
  Color color = new Color();
  Node (int _id) { id = _id; }
  void print() {
    if (Conf.COLORED)
      Color.setDisplayColor(color);
    System.out.print(id);
  }
}


class Edge {
  Node a, b;
  Color color = new Color();
  Weight weight;
  Edge(Node _a, Node _b) {a=_a; b=_b;}
  void print() {
    if (Conf.COLORED)
      Color.setDisplayColor(color);
    System.out.print(" (");
    a.print();
    System.out.print(" , ");
    b.print();
    System.out.print(") ");
    if (Conf.WEIGHTED) weight.print();
  }
}


class Color {
  static void setDisplayColor(Color c)...
}
class Weight {
  void print() { ... }
}
```

- Choices read from command line and stored in `Conf`.
- Other classes check variables and invoke code appropriately.

Discussion

- Variation is evaluated at run-time.
- All functionality is included, even if never used.
  - More memory required.
  - If-statements add computational overhead.
  - Security risks introduced, i.e., buffer overflow attacks.

```
Edge add(Node n, Node m, Weight w) {
  if (!Conf.WEIGHTED)
    throw new RuntimeException();
  Edge e = new Edge(n, m);
  e.weight = w;
  nodes.add(n);
  nodes.add(m);
  edges.add(e);
  return e;
}
```

Discussion

- Can alter feature selection at run-time.
  - However, code may depend on initialization steps.
  - May be easier to restart.
- Can pass to methods instead of setting globally.
  - Allows different configurations throughout program.

```
Edge add(Node n, Node m, Weight w) {
  if (!Conf.WEIGHTED)
    throw new RuntimeException();
  Edge e = new Edge(n, m);
  e.weight = w;
  nodes.add(n);
  nodes.add(m);
  edges.add(e);
  return e;
}
```

Discussion

- Conditional statements are a form of annotation.
  - Mark boundaries between features.
- Global variables reduce independence of modules.
  - However, passing many arguments reduces understandability/requires repetition.
  - Pass a “configuration object” containing settings.
- Feature code mixed and scattered across project.
  - Hard to understand and change.

Benefits and Drawbacks

- Benefits
  - Easy to understand and use.
  - Flexible
  - Allows different configurations in same program.
- Drawbacks
  - All code in executable.
  - Feature code and configuration knowledge scattered across program.
  - Difficult to link feature model and implementation.

We Have Learned

- *How* do we build a custom product from a feature selection?
  - Binding Time
    - Compile, load, run-time
  - Technology
    - Language vs Tool-Based Implementation
  - Representation
    - Annotation vs Composition

We Have Learned

- Preprocessors
  - Mark code to include in compiled executable.
  - Omit code that we do not select entirely.
  - Compile-Time, Tool-Based, Annotation-Based
- Build Systems
  - Replace files based on feature selection.
  - Compiler options set using features.
  - Compile-Time, Tool-Based, Annotation-Based

We Have Learned

- Parameters
  - Set Boolean variables via command-line, config file, GUI, API, etc. globally or pass to methods.
  - Use if-statements to execute correct code.
  - Load or Run-Time, Language-Based, Annotation-Based

Next Time

- Guest lecture - Henrik Lönn, Volvo Trucks
  - Product lines and feature modelling in industrial development.
  - **On Zoom - NOT in person**
  - Will be on individual assignment, so attend!
- Assignment 2 - any questions?
  - Due November 21
  - Feature modelling and analysis for mobile robots

UNIVERSITY OF GOTHENBURG

CHALMERS

UNIVERSITY OF TECHNOLOGY